

# The arrow logic of meta-specifications: a formalized graph-based framework for structuring schema repositories\*

Zinovy Diskin

Lab. for Database Design  
*Frame Inform Systems*, Ltd.  
Riga, Latvia  
Diskin@aol.com

## Abstract

A graphical formalized language is proposed for structuring specification (schema) repositories. The language is based on the notion of mapping, or morphism (arrow), between schemas and suitable for any kind of specifications for which morphisms are defined. Specifically, it is shown that notions of view, refinement and schema integration can be precisely formulated in this arrow framework and correspondingly specified. Basic constructs of the language can be considered as specialization of very general constructions developed in mathematical category theory. This provides the language with a firm mathematical foundation.

## 1 Introduction and preliminary discussion

It is clear that a working description of large, and of moderate size too, business/information system cannot be a flat, one-piece, specification. Rather, it will be a collection of different viewpoint specifications and their refinements of different degree of elaborating details. This collection appears as a set of basic specifications (level 0), their views and refinements (level 1), views and refinements (of level 2) of these 1-views and 1-refinements and so on. In addition, the components (views and refinements) can be somehow additionally related between themselves, for example, one component specification can be the intersection of several others, or the merge of some others, or both. Thus, specification of complex system is itself a complex structure subjected to certain integrity constraints and carrying

---

\*Supported by Grants 93.315 and 96.0316 from the Latvian Council of Science.

certain operations, and so to be manageable, it must be described in some precise way. To fix terminology, we will call component specifications *schemas* and the entire specification repository *schema library*.

Managing structures as above needs a suitable language to arrange these structures for some predefined meta-specification pattern. Several proposals for such languages/patterns were made in research papers and implemented in CASE-tools. It is common to arrange them around the structuring primitives of view to a schema, refinement of a schema and schema integration. In particular, in a series of works by the Italian school of conceptual modeling ([3, 18, 11]), a general mechanism of *view/refinement grid* was proposed.

However, their grid and similar patterns that can be found in the literature lack a construct of vital importance for the problem: namely, what is a structuring pattern for specifying relations between schemas? In other words, given two schemas  $S_1$  and  $S_2$ , how can one specify that  $S_2$  is a view on  $S_1$  or, for example, that a third schema  $S$  is the result of integration of  $S_1$  and  $S_2$ ? Similarly, how can one specify that a schema  $S_2$  is a refinement of schema  $S_1$ ?

The goal of the present paper is to outline a general language in which the notions of view and refinement, and of operation over schemas, *eg*, integration, can be precisely specified in some abstract unified way. *Abstract* here means that our definitions will be applicable for any kind of schemas, for example, ER-diagrams of some predefined kind, or business specifications of some predefined kind, or, say, Goguen's sign systems [12]. Of course, for applicability to each particular kind of schemas, say, SPEC, one has to have precisely defined what is a SPEC-schema, and, in addition, what is:

- **SPEC-schema mapping (morphism)**,  $f: S_1 \rightarrow S_2$ . Informally, schema morphisms can be thought of as mappings sending items of the source schema  $S_1$  into items of the target schema  $S_2$  such that the structural relationships between schema items described in the definition of SPEC are respected. For example, if schemas are object class/reference schemas, then schema mappings send class names into class names and references into references in such a way that if  $r$  is a reference of objects of some class  $C$  to objects of class  $D$  in schema  $S_1$  then  $f(r)$  is a reference of  $f(C)$ -objects to  $f(D)$ -objects in schema  $S_2$ .
- **Derived information operation over schemas**. These DER-operations augment SPEC-schemas with new items to be thought of as derived from basic items. Given a SPEC-schema  $S$ , various augmentations  $\overline{S}$  of  $S$  can be built by consecutive applications of DER-operations to  $S$ , actually one can build various chains  $S \subset \overline{S'} \subset \overline{S''} \dots$  in function of which DER-operations, and in what order, are applied. For example, if SPEC-schemas are database schemas of some kind, DER-operations are nothing but queries: a query  $q$  against  $S$  is specified by a schema  $S_q$  so that the schema  $S \cup S_q$  specifies  $S$ -data augmented with the answer to the query. A collection of queries against  $S$  is then specified by some schema  $\overline{S}$  containing  $S$ . For another example, if SPEC-schemas are Goguen's sign systems [12], then DER-operations are sign constructors described in [12].
- **Schema diagram predicate**. It is a property of some diagram consisting of

schemas and their mappings. A special case of diagram predicate is a *diagram operation* like, *eg*, schema integration (this is quite similar to, say, presenting binary operation of adding integers by the corresponding ternary predicate).

In addition, some not so evident on the informal level but technically as important as above is the condition of having defined the construct of

- **Derived schema mapping augmentation.** That is, given a mapping  $f: S_1 \rightarrow S_2$  and schema augmentation  $\overline{S}_1 \supset S_1$ , an augmented mapping  $\overline{f}: \overline{S}_1 \rightarrow \overline{S}_2$  must be determined in a unique way, where  $\overline{S}_2$  is the augmentation of  $S_2$  similar to  $\overline{S}_1$  (built with the same queries in the database example) . Of course, the restriction of  $\overline{f}$  on the schema  $S$  coincides with  $f$ ,  $\overline{f} \upharpoonright_S = f$ .

In this language, the following definitions (motivated and exemplified below in the paper) make sense.

**1.1 Views and refinements: Definition.** Let  $S$  be a schema (specification) of some predefined kind.

(i) A *view* to  $S$  is a pair  $V = (S_V, v)$  with  $S_V$  a schema of the same kind as  $S$ , the *view schema*, and  $v: S_V \rightarrow \overline{S}$  is a schema mapping, the *view morphism*, into some augmentation of  $S$  with derived items.

(ii) A *refinement* of  $S$  is a pair  $R = (S_R, r)$  with  $S_R$  a schema of the same kind as  $S$ , the *refinement schema*, and  $r: S \rightarrow \overline{S}_R$  is a schema mapping, the *refinement morphism*, into some augmentation of  $S_R$  with derived items.  $\diamond$

As for schema integration, it was shown in [4, 5] that it can be specified in the following way.

**1.2 Schema integration: Definition.** *Integration* of schemas  $S_1, S_2$  according to some *correspondence information* is a special arrow operation specified by the diagram

$$\begin{array}{ccc}
 \overline{S}_1 & \xleftarrow{p_1} & S_{\text{Corr}} \\
 q_1 \downarrow & \langle \text{Int} \rangle & \downarrow p_2 \\
 \overline{S}_{\text{Int}} & \xleftarrow{q_2} & \overline{S}_2
 \end{array}$$

where  $(\overline{S}_1, \overline{S}_2, S_{\text{Corr}}, p_1, p_2)$  specifies the input and  $(\overline{S}_{\text{Int}}, q_1, q_2)$  the output of the operation.

The schema  $S_{\text{Corr}}$  describes some common part of  $S_1$  and  $S_2$  so that the triple  $(S_{\text{Corr}}, p_1, p_2)$  specifies some relation or correspondence between  $S_1$  and  $S_2$ .  $\overline{S}_{\text{Int}}$  is the integrated schema and the mappings  $q_1, q_2$  show how the component schemas are presented in the integrated one.  $\langle \text{Int} \rangle$  is a label (operation marker) denoting some procedure of integration (see [5] for details of this description).  $\diamond$

**1.3 Meta-specifications via arrows: Discussion.** The framework we discuss is based on arrows and needs the corresponding arrow machinery, and even special arrow thinking. Such a machinery, and way of thinking, were developed in mathematical category theory<sup>1</sup>

<sup>1</sup>a standard reference suitable for computer science is [2]

and have proven their extreme effectiveness in studying various logical systems, in particular, logics employed in computer science and artificial intelligence (see also an application in [7]). In particular, morphisms of the kind  $f: S_1 \rightarrow \overline{S_2}$  are well studied in category theory under the name of *Kleisly morphisms*<sup>2</sup>. Also, the arrow framework reveals a remarkable duality between two infamous problems: view updates and lossless (preserving information capacity) refinements, we will outline this very briefly in section 3.3.

As for schema integration, the pattern stated by definition 1.2 is of quite general nature: in category theory couples like  $\langle S_{\text{CORR}}, [p_i, i = 1, \dots, n] \rangle$  are called *n*-ary relations because in a majority of situations they behave very much like ordinary relations over the targets of mappings  $p_i$ . For example, when schemas are just plain sets,  $S_{\text{CORR}}$  can be considered a set of *n*-tuples and  $p_i$  are projections. In addition, if schemas are organized into some category in the sense of category theory, then schema integration turns out to be a well-known categorical operation – the so-called *push-out*. In fact, definition 1.2 is nothing but the general shape of the push-out operation described in schema integration terms.

So, there is nothing *ad hoc* in the general framework above and, on the other hand, despite its abstract character this framework suggests quite instrumental approaches to the problems in question: for schema integration it was shown in [4, 5], and as for structuring schema libraries, it will be shown in the present paper.

It is worth noting in this respect that the entire framework we discuss is nothing but a general arrow treatment of the old issue of mathematical logic on logical theories and their interpretations. Indeed, in a sense, theories and specifications are synonyms and specification mappings are nothing but theory interpretations when basic primitives of one theory are interpreted by, maybe, derived primitives of another theory, for example, multiplication of integers is interpreted by their addition. This issue is well studied in the classical setting of one-sorted and multi-sorted theories but interpretation of theories with different domain structures of their models when interpretation involves operations on domains – just the setting with business specifications – is an open problem. A natural, and practically without alternatives, approach to the problem is to use general arrow patterns of categorical logic but even here the problem is still a challenge.

An important benefit of using our framework is that it is graphical, and hence is much more manageable by humans. However, in contrast to many other graphic languages, the arrow language we propose is quite precise and possesses a formal semantics, thus, our meta-specifications are free of ambiguities and directly suitable for computer implementation.  $\diamond$

**1.4 Sketches as component specifications: Discussion.** As it was stressed above, an important advantage of the arrow framework is that it is polymorphic, that is, is suitable for any kind of specifications. One must only define what are schemas and their morphisms (form a category of schemas, as a category theorist would say), and set up the augmentation mechanism outlined above. In particular, the framework can be applied to

---

<sup>2</sup>actually, in a slight different setting when instead of set of DER-operations the corresponding operator of closing a schema with all possible derived items is considered (this operator is called the *monad* generated by the set DER)

repositories of *sketch* specifications, the latter were described in [9, 8, 7] but to make the paper selfcontained, a brief outline of sketches is presented in Appendix. In particular, the arrow diagram in definition 1.2 is nothing but a simple sketch – a directed multigraph in which some diagrams are marked with labels taken from a predefined signature – and this sketch is used to specify a meta-specification construct.

So, in the paper sketches will be used in the two ways:

(i) in meta-specifications as in definition 1.2, nodes of such sketches denote schemas (specifications) and arrows are mappings between them;

(ii) as sample specifications, that is, to exemplify and demonstrate the essence of the arrow approach to meta-modeling, we will also use sketches as a running example of schemas, say, for data modeling: nodes of such sketches can be thought of as denoting object classes and arrows denote references between them.

The reason for choosing sketches as a running schema example is that in the arrow approach morphisms are more important than objects<sup>3</sup>, and the sketch mapping is a well defined notion. In contrast, for a majority of specification languages used in software engineering, the notion of mapping between specifications usually is not defined. Moreover, an attempt to define specification mappings often reveals some shortcomings in the definition of specifications as such. In fact, such a situation is well known in category theory where well definability of mappings between constructs of some new kind to be defined is used as a test against the construct definition in question. In fact, the slogan underlying arrow thinking is that to *objectify* means to *mappify* and rich experience of category theory in specifying mathematical constructs, and in applications to computer science as well, justify and exemplify the slogan (see also [13]).

A good example of the general phenomenon above can be found in the well known ER-diagrams. There are several precise definitions of ER-diagrams of one or another kind but no correct definitions of their mappings can be find in the literature, and this is not accidental. Indeed, a natural ER-diagram mapping should send rectangle nodes (entities) to rectangle nodes, and diamond nodes (relationships) to diamond nodes. However, in such a setting the well known, and practically important, phenomenon of semantic relativism is ruled out. If, on the other hand, one allows interpreting rectangles by diamonds and vice versa, then the crucial for ER data modeling distinction between entities and relationships will disappear. A proper generalization of the ER-model where this problem can be resolved was proposed in [9], and this "arrow-based ER" turned out nothing but the sketch data model, that is, an application of the general sketch specification pattern to data modeling. In addition, in the sketch framework the widely discussed opposition of ER and OO disappears, and thus the infamous problem "ER-vs-OO" turns out to be pseudo-problem caused by using in the discussion improper languages rather than by the essence of the phenomenon (see [8] for details).

A general conjecture one can make in this respect is that as soon as one refines some graphic notation heuristically invented in a problem domain, in particular, provides it with a formal semantics and makes it arrow-compatible, one will necessarily come to one or another kind of sketches (cf. [6, 10]). Thus, using sketches as a running example of

---

<sup>3</sup>more accurately, all properties of objects are defined through arrows, that is, through arrow interfaces to objects

component schemas can be considered in a much wider context of sketches-as-prototypical-specifications.  $\diamond$

So, the main proposal of the present paper is to specify a schema library by a (meta-)sketch whose nodes denote component (basic, view and refinement) schemas and arrows denote (view and refinement) mappings between them. More precisely, the construction can be thought of as a collections of planes (fibers), each is a sketch of view schemas and mappings, which are connected by inter-plane refinement arrows. This provides a consistent modularization mechanism having both a substantial semantics – arrows are views and refinements, and formal semantics – each item in meta-sketches has a precise formal meaning.

The entire construction may look monstrous but in category theory the corresponding notation, terminology and even techniques of presenting these arrow towers on a flat paper were elaborated. In effect, the arrow machinery provides a clear and handy framework for structuring schema repositories and can be useful in specifying and design of complex business systems. What should be stressed in this respect is that abstract categorical patterns turned out to be unexpectedly close to constructs appearing in design practice. In particular, *schema grids*, a general pattern for structuring schema libraries introduced in [3], turns out to be a very special case of our *metasketch fibrations* (some details are at the end of section 4).

The rest of the paper is organized as follows. In Appendix foundations of the sketch data model are very briefly described, more detailed presentation can be found in [7]. In section 2 the arrow setting for views, and in section 3 – for refinements are described. Section 4 is the culmination, a general graph-based framework for structuring schema libraries is presented there.

**Acknowledgement.** I am indebted to Haim Kilov for general encouragement on applying arrows to business specifications and, in particular, for several helpful suggestions on the material of this paper and its presentation as well.

## 2 Views via schema morphisms

The notion of view is one of the most important for modularizing complex specifications. It is important also technically since makes it possible to provide each application with its own representation of the entire universe and isolate it from inessential (for this application) details and changes in the base specification. In fact, views appear in different context and under different names in a wide range of intellectual activities. As for software engineering, the view construct had been seriously studied, and was technically realized, in the database theory and practice.

**2.1 Getting definition.** There was a considerable debate in the database theory literature about the general notion of view against a database schema, in particular, in the object-orientation framework. Despite the large body of work done in the area ([1, 20, 19, 17] to mention few), the notion of view was not precisely formulated and even the terminology is still somewhat confusing. Indeed, in the majority of works (like

that just mentioned) the term *view* is used to denote derived items (as a rule, classes or relations), according to the pattern “**define view name as query specification**”. In some works, however ([21, 14]), a *view* to a schema is a subschema of the schema. An evident integrated formulation is to consider a view to a schema  $S$  as a subschema of some augmentation  $\overline{S}$  of  $S$  with derived items, *ie*,  $S_{View} \subset \overline{S} \supset S$ . This is an almost good definition but it forces one to consider the name set of view schema  $S_V$  as a subset of that of the schema  $S$ . However, for the view user its schema  $S_V$  should be totally autonomous with its own name set.

Thus, a more correct consideration is to define a *view* over  $S$  as a pair  $V = (S_V, v)$  with  $S_V$  a schema (of the same kind that  $S$  is) and  $v$  a mapping (schema morphism)  $v: S_V \rightarrow \overline{S}$  sending items of  $S_V$  into those of  $\overline{S}$ . In this way one comes to the general definition 1.1 where, as it was stressed above, schemas are specifications of some arbitrary but predefined kind. For example, one may think of them as object class-reference schemas or sketches and of arrows as mappings sending items of source schemas into those in target schemas.

An essential advantage of this definition is that the mapping  $v$  is not bound to be one-one, that is, it can well be the case when two different items  $N, M$  in  $S_V$  are glued into one item  $K$  in  $S_V$ ,  $v(N) = v(M) = K$ . Then the presence of two different names in  $S_V$  could seem a trick that just misleads the view user, but note that the schema  $S$  may evolve and later the item  $K$  may diverge into two different items  $K_1, K_2$  such that according to the new view semantics the view morphism  $v$  has to map  $N$  to  $K_1$  and  $M$  to  $K_2$ ,  $v(N) = K_1 \neq K_2 = v(M)$ . Or, vice versa, initially there were two different nodes  $K_1, K_2$  with  $v(N) = K_1, v(M) = K_2$  but then  $S$  has evolved to a state when  $K_1$  should be merged with  $K_2$ . Thus, by means of a suitable changing the view mapping  $v$  but *without changing* the view schema  $S_V$ , one can conform an application built over the view with evolving entire specification  $S$  so that there is no need to rebuild the application.

◇

**2.2 Examples.** Let us consider a simple conceptual schema, sketch  $S$ , on Fig. 4 in Appendix. Two simple views  $V_1, V_2$  on this sketch are presented on Fig. 1(b) on the left. Each of the views is specified by the view schema and a mapping which is set by the corresponding table; an augmentation of  $S$  required to set up these views is specified on Fig. 1(a) and described in section A.3 of Appendix. From these specifications it is seen, for example, that extension of the node  $PHusband$  in the view  $V_1$  consists of those  $S.Man$ -objects whose *income* is more than  $10^6$  and, simultaneously, they are not present in the relation  $\overline{S}.CurrMarried^{*4}$ . Indeed, the markers  $\langle Im \rangle'$  and  $\langle Dif \rangle^\sim$  on Fig. 1(a) denote operations of taking the image of function and set difference respectively. Then the class  $Man^\sim$  consists of those  $Man$ -objects which do not occur into  $CurrMarried^*$ -pairs. Further, the marker  $\langle CoIm \rangle^?$  defines the class  $Man^?$  as the intersection of  $Man^\sim$  and  $Man^{\$}$ . The latter class is defined by the marker  $\langle CoIm \rangle^{\$}$ , that is, it consists of  $Man$ -objects with *income* greater than  $10^6$  (so, extension of the node  $Man^?$  consists of rich unmarried men). According to the  $V_1$ -view mapping, the extension of the node  $S_{V_1}.PHusband$  is as was described above.

---

<sup>4</sup>we will qualify sketch items by names of sketches if necessary

Similarly, extension of the node *Marr* in the view  $V_2$  consists of those  $\overline{S}.CurrMarried^*$ -pairs  $(w, h)$  for which

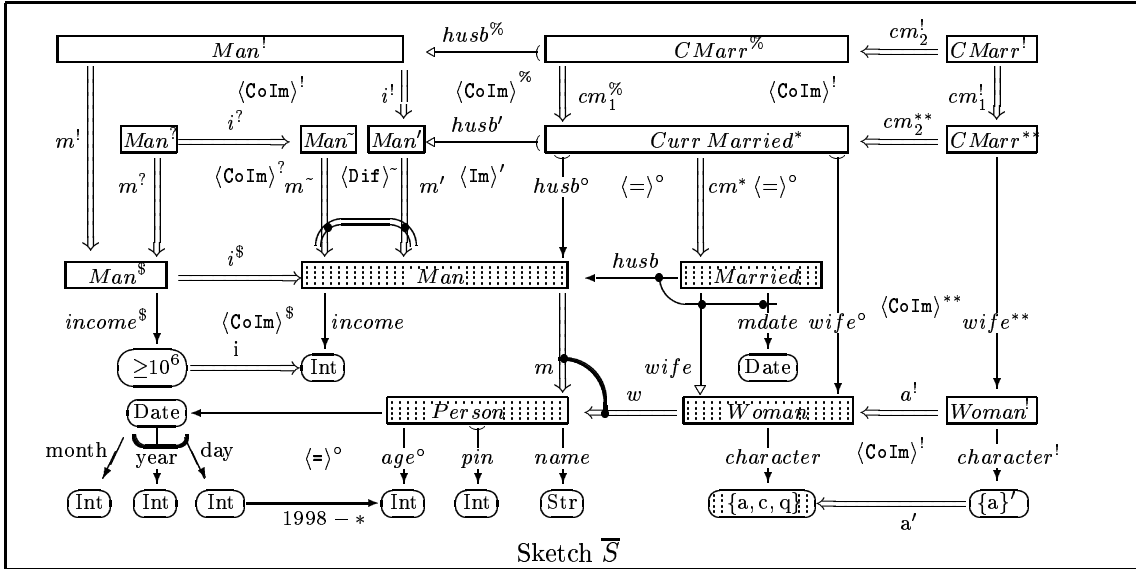
$$w.character = a \text{ and } h.income \geq 10^6.$$

Some delicate points associated with our definition of view are demonstrated in the right half of Fig. 1(b). A view mapping  $v$  must be a schema morphism. That is, for the sketch data model,  $v$  is not merely a graph morphism but a sketch morphism: if a diagram  $D$  in  $S_V$  is labeled by a marker  $M$  then its image  $v(D)$  in  $S$  must be also labeled by  $M$ . For example, in the specification  $V_3$  on Fig. 1(b) the corresponding mapping is not a sketch morphism since the pair  $(S_{V_3}.husb, S_{V_3}.wife)$  is marked by an arc while its image, the pair  $(\overline{S}.husb, \overline{S}.wife)$  in the sketch  $\overline{S}$  is not labeled (well, the triple  $[\overline{S}.husb, \overline{S}.wife, \overline{S}.mdate]$  is separating but it does not imply that the pair is). Semantically, this means that according to the view schema  $S_{V_3}$ , any *Married*-object is identified by the pair of objects  $(wife, husb)$  whereas the data which the view extracts from the  $S$ -extension do not necessarily satisfy this condition. Hence,  $V_3$  is not a view on  $S$  in the technical sense.

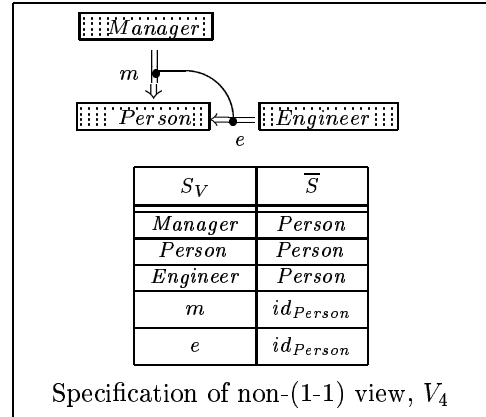
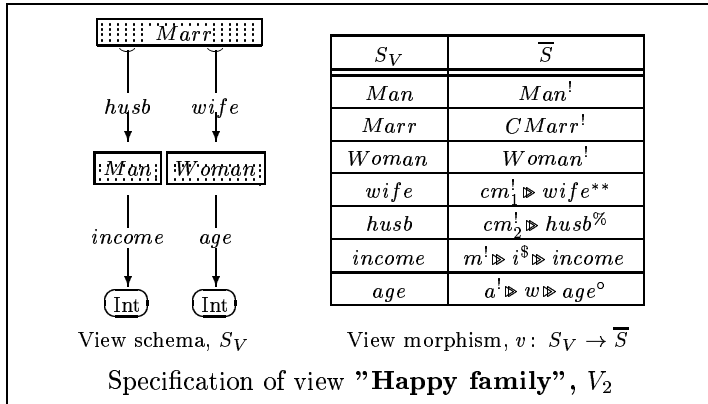
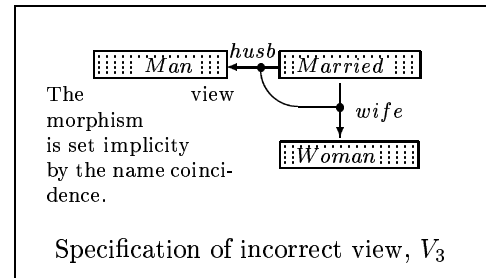
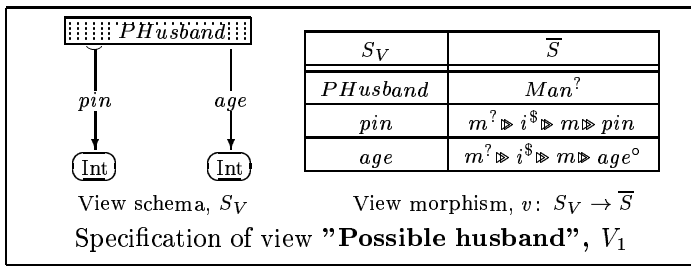
The view  $V_4$  is a somewhat artificial example of view where the view mapping is not one-one. All the three nodes in  $S_{V_4}$  are mapped into the same node  $S.Person$  and both *IsA*-arrows,  $m, e$  are mapped into the arrow  $S.id_{Person}: S.Person \rightarrow S.Person$ , we assume that every node in a sketch has, by default, the identity arrow into itself, which is not depicted. For the present sketch  $S$  the view  $V_4$  has no much sense but suppose that a more detailed schema of the universe will be built where subclasses *Manager* and *Engineer* of the class *Person* will appear. Then we will have the possibility to change mapping  $v_4$  in the evident way to adjust view  $V_4$  to a new universe schema without changing the view schema  $S_{V_4}$ .  $\diamond$

**2.3 View metasketch.** The system of views we have just considered can be presented by a graph on Fig. 1(c). Moreover, this graph carries a sketch structure since view mappings and their diagrams are subjected to certain constraints. For example, the arc with brackets hung on the arrows  $v_1, v_2$  denotes the constraint that views  $V_1, V_2$  are disjoint. Also, tail arcs on arrows  $v_1, v_2$  show that views  $V_1, V_2$  must be one-one (see Table 1). One can see that specifying the sketch structure in the graph of views is important for capturing meta-specification semantics.  $\diamond$

**2.4 Semantics.** The crucial observation is that semantics of schemas can be also described by arrows. For example, if one thinks of schema as a sketch modeling some data, then extension of the schema is a mapping that sends nodes to sets and arrows to functions (built over some predefined universe of objects,  $\mathcal{U}$ ) in such a way that intended semantics of the markers is respected. One can consider the collection of  $\mathcal{U}$ -sets and  $\mathcal{U}$ -functions as a graph whose nodes *are* sets and arrows *are* functions, then extension mapping is a graph morphism. In addition, given a marker (predicate)  $m$ , those diagrams of  $\mathcal{U}$ -sets and functions that possess the property denoted by  $m$  can be thought of as *marked* by  $m$ . In this way the universe can be converted into a (monstrous) schema  $U$  of the same type as the schemas we consider, and then extension of  $S$  is nothing but a schema morphism  $e: S \rightarrow U$ .



a) Composition of queries against the sketch  $S$  on Fig. 4



b) Several simple views to the sketch  $S$  on Fig. 4

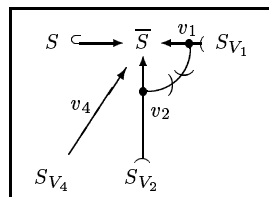


Figure 1: Specifying views over a sketch

If  $\overline{S}$  is an augmentation of  $S$  with derived items, any  $S$ -extension  $e: S \rightarrow U$  can be extended to an extension of the augmented schema,  $\overline{e}: \overline{S} \rightarrow U$ , in a unique way as follows. If an item  $N$  from  $\overline{S}$  actually occurs into  $S$ ,  $N \in S$ , then  $\overline{e}(N) = e(N)$ . If  $N \in \overline{S} \setminus S$  then  $N$  occurs into the schema of some query  $q$  against  $S$  and so  $\overline{e}(N)$  is the answer to  $q$  extractable from the set of data  $\{e(M) \mid M \in S\}$ .

Now, given a view  $v: S_V \rightarrow \overline{S}$  over a schema  $S$  and an extension  $e$  of  $S$ , the corresponding extension  $e_V$  of the view schema is the composition of two arrows:

$$e_V = v \triangleright \overline{e}: S_V \xrightarrow{v} \overline{S} \xrightarrow{\overline{e}} U$$

(the symbol  $\triangleright$  denotes composition of two arrows for which the target of the left arrow coincides with the source of the right one). Moreover, if  $v_1: S_{V_1} \rightarrow \overline{S_V}$  is a view over  $S_V$ , then the extension  $e_{V_1}$  of  $S_{V_1}$  is the composition

$$e_{V_1} = v_1 \triangleright \overline{v} \triangleright \overline{e}: S_{V_1} \xrightarrow{v_1} \overline{S_V} \xrightarrow{\overline{v}} \overline{S} \xrightarrow{\overline{e}} U$$

where  $\overline{v}: \overline{S_V} \rightarrow \overline{S}$ ,  $\overline{e}: \overline{S} \rightarrow U$  are the corresponding augmented mappings.

Thus, a system of views over a given schema is specified by a graph of view schemas and view mappings, and the well known mechanism of calculating view extension is specified by the arrow composition.  $\diamond$

Let us designate the set of all possible semantic extensions of  $S$  by  $\mathbf{Ext}(S)$ , and so  $\mathbf{Ext}(S) = \mathbf{Arr}(S, U)$  – a standard categorical notation for the set of all morphisms between two objects.

With any view  $v: S_V \rightarrow \overline{S}$  there is correlated a mapping

$$v^*: \mathbf{Ext}(S) \rightarrow \mathbf{Ext}(S_V), v^*(e) \stackrel{\text{def}}{=} v \triangleright \overline{e}$$

resulted from the derivation of view extension from the schema extension as we have just explained. Many important properties of view  $v: S_V \rightarrow \overline{S}$  are determined by the properties of the mapping  $v^*$ . In particular, view updates are easily managed if  $v^*$  is injective (one-one).

### 3 Refinements via schema morphisms

**3.1 Definition.** As it was described in introduction, a refinement of a schema (sketch)  $S$  can be thought of as schema morphism  $r: S \rightarrow \overline{S_R}$  into an augmentation of the refining schema with derived items.

Normally, the image of  $r$  should contain derived items: say, for some item  $B \in S$  there are items  $D_1, \dots, D_n \in S_R$  such that  $r(B) = \mathbf{Op}(D_1, \dots, D_n)$ . That is,  $r(B) = D$  where  $D$  is the result of some operation (query) against  $S_R$ ,  $\mathbf{Op}$ , whose input consists of items  $D_1, \dots, D_n$ ,  $D \stackrel{\text{def}}{=} \mathbf{Op}(D_1, \dots, D_n)$ . Then one could say that  $B$  is *implemented* by  $\mathbf{Op}$  and  $D_1, \dots, D_n$ .

Similarly to view systems, the machinery of morphism augmentation and composition provides explanation of refinement chains: if  $r_1: S \rightarrow \overline{S_{R1}}$  is a refinement of  $S$  and  $r_2: S_{R1} \rightarrow \overline{S_{R2}}$  is a refinement of  $S_{R1}$ , then the composition

$$r_1 \triangleright \overline{r_2}: S \xrightarrow{r_1} \overline{S_{R1}} \xrightarrow{\overline{r_2}} \overline{\overline{S_{R2}}}$$

is also a refinement of the initial schema  $S$ .  $\diamond$

**3.2 Example.** A simple example is presented on Fig. 2. The refinement mappings  $r_1, r_2$  are specified by the two-row tables. From table  $r_1$  it can be seen, for example, that the node  $S.Vehicle$  in schema  $S$  is refined by the disjoint union of nodes  $S_1.OffVehicle$  and  $S_1.PrivVehicle$  in schema  $S_1$  (*Off* stands for *Office* and *Priv* for *Private*), note the marker  $\langle \text{II}' \rangle$  on the corresponding diagram. The relation  $S.Assignment$  is refined by the disjoint union of the relations  $S_1.Ass - ment^*$  and  $S_1.Ownership$ , this is specified by the marker  $\langle \text{II}'' \rangle$ . The latter relation is a basic item in the schema  $S_1$  while  $S_1.Ass - ment^*$  is a derived one: it is the graph of the function  $S_1.driven\_by$ , which is shown by the marker  $\langle \text{Gra} \rangle^*$  hung on the corresponding diagram.

The refinement  $r_2$  can be traced in a similar way (see the corresponding table). Class  $S_1.Person$  is refined as the union of classes  $S_2.Student$  and  $S_2.Staff$  (basic for schema  $S_2$ ) whose intersection is given by class  $S_2.EmplStudent$ . Extension of the latter is calculated via key attributes  $\#$ 's of operand classes as it can be seen from the marker  $\langle \text{Join} \rangle^*$  hung on the corresponding diagram. After this, extension of the node  $S_2.Faculty$  is calculated as specified by the union operation denoted by marker  $\langle \text{U} \rangle^{**}$ . Similarly, node  $S_1.PrivVehicle$  is refined by the union of  $S_2.Bicycle$  and  $S_2.PrivCare$ , while node  $S_1.OffVehicle$  is just renamed into  $S_2.DepCare$ . Arrow  $S_3.vowns^!$  is derived as union of arrows *bowns* and *cowns* (consider arrows as graphs – sets of pairs). Then, node  $S_3.Ownership^{\%}$  together with arrows  $pv^{\%}$ ,  $st^{\%}$  are obtained as the graph of arrow  $vowns^!$  (note the corresponding marker  $\langle \text{Gra} \rangle^{\%}$ ).  $\diamond$

**3.3 Semantics.** The entire mechanism is similar to that of views but works in the reverse direction through the function  $r^*: \mathbf{Ext}(S_R) \rightarrow \mathbf{Ext}(S)$ ,  $r^*(e') \stackrel{\text{def}}{=} r \triangleright \overline{e'}$  for any  $e' \in \mathbf{Ext}(S_R)$  (cf. with mapping  $v^*$  described at the end of section 2.4).

There is a remarkable (algebraic) duality of the two infamous problems: view updates and lossless (preserving information capacity) refinements when they are formulated in our framework. Indeed, while for view updatability the key problem is in injectivity of  $v^*$ , for lossless refinements the key is in surjectivity (the cover property) of  $r^*$ : mutual duality of injective and surjective mappings is well known in category theory. The premise of this duality can be seen immediately from the main definition 1.1: the pattern for views is  $*$ :  $\circ \rightarrow \bullet$  while that for refinements is  $*$ :  $\bullet \leftarrow \circ$  where  $\bullet$  denotes location of the base schema  $S$  and  $\circ$  denote location of the derivative view/refinement schema  $S_V/S_R$ .  $\diamond$

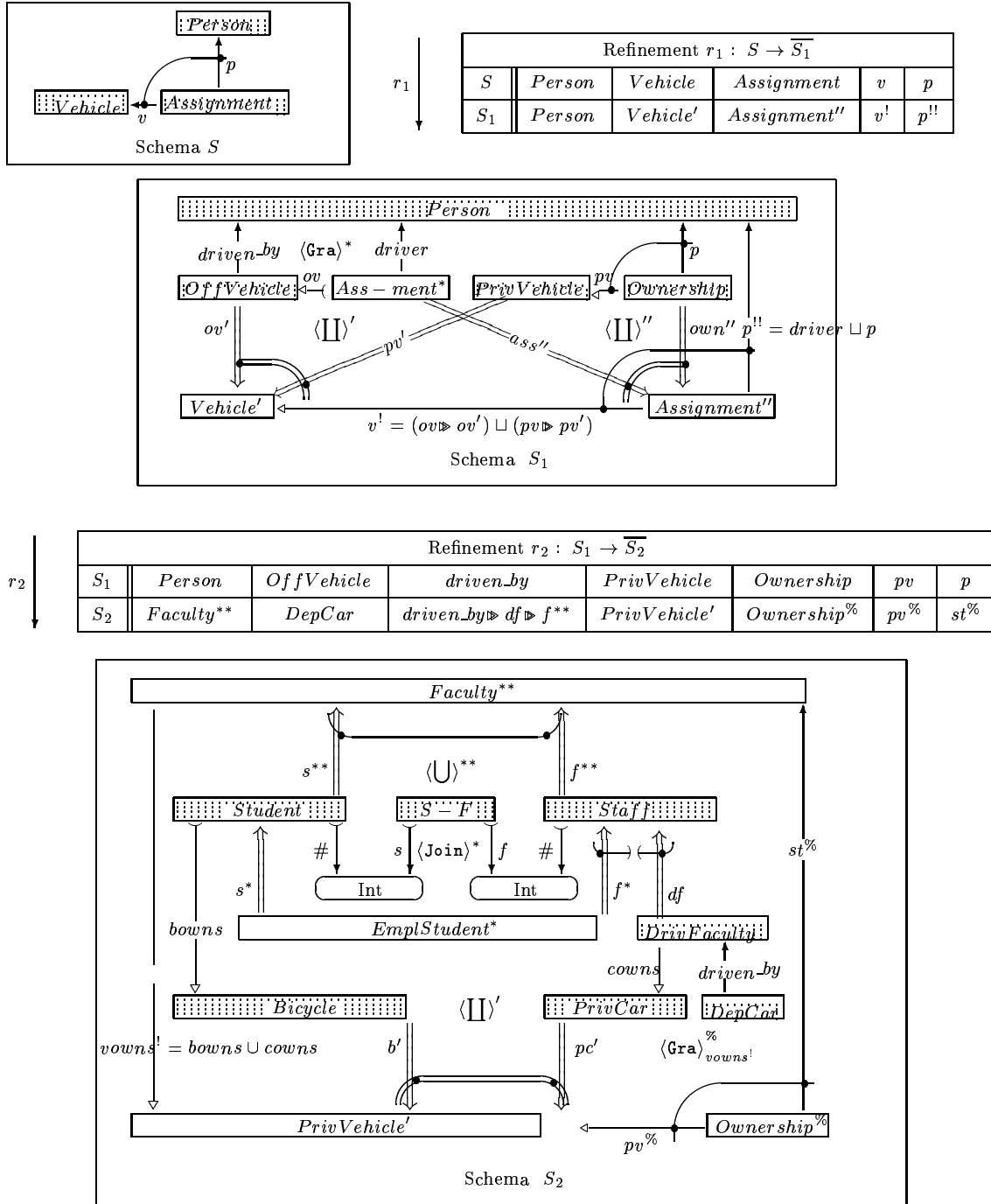


Figure 2: Refinement via sketch morphisms

## 4 Hypersketch fibrations vs. specification repositories

**4.1** The top-down design methodology suggests to specify complex system by the process of stepwise specification refinements, and in each stage of refinement to work out a system of user views to principal schemas. Such a design process can be specified by a chain

$$\mathcal{S}_1 \overset{\mathbf{r}_1}{\rightsquigarrow} \mathcal{S}_2 \overset{\mathbf{r}_2}{\rightsquigarrow} \dots \overset{\mathbf{r}_{n-1}}{\rightsquigarrow} \mathcal{S}_n$$

where each  $\mathcal{S}_i$  is a *view metasketch* on the  $i$ -th stage of refinement and  $\mathbf{r}_i$  is the corresponding refinement. Here *view metasketch* means a sketch whose nodes denote schemas and arrows denote mappings between them (views). In other words, with each metasketch  $\mathcal{S}_i$  there is coupled a mapping (functor)  $\mathcal{E}_i$  sending nodes and arrows of  $\mathcal{S}_i$  into schemas and schema mappings. By analogy with hypertexts, we will call such a construct *hypersketch*.

Each arrow  $\mathbf{r}_i$  is a pair  $(\mathcal{F}_i, \rho_i)$  where the first component is a mapping of sketches,  $\mathcal{F}_i: \mathcal{S}_i \rightarrow \mathcal{S}_{i+1}$ , and the second component is a function which assigns to any view schema  $S \in \mathcal{S}_i$  a refinement mapping  $\rho_i^S: S \rightarrow \overline{\mathcal{F}_i(S)}$ . One can think of this construction as a collection of view hypersketches, each is placed in its own fiber (plane) indexed by the number of refinement steps, while  $\mathbf{r}_i$ 's are inter-fiber mappings. This explains the name *hypersketch fibration* we use, the term *fibration* is borrowed from the category theory.

In addition, the following diagram must hold commutative for any  $i$  and any view mapping  $v: S_V \rightarrow \overline{S}$  in  $\mathcal{S}_i$ :

$$\begin{array}{ccc} S_V & \xrightarrow{\rho_i^{S_V}} & \overline{\mathcal{F}_i(S_V)} \\ v \downarrow & & \downarrow \mathcal{F}_i(v) \\ \overline{S} & \xrightarrow{\rho_i^{\overline{S}}} & \overline{\mathcal{F}_i(\overline{S})}. \end{array}$$

In other words, refinement of a view on a schema is the corresponding view on the schema refinement.  $\diamond$

**4.2** The commutativity condition above is an important constraint: its maintenance is necessary for holding integrity of the schema repository and should be a mandatory functionality of any meta-specification framework. On the other hand, this condition makes the pair  $(\mathcal{F}_i, \rho_i)$  above a construction which has been thoroughly studied in category theory: namely, with this condition the mapping  $\rho_i$  becomes nothing but a so called *natural transformation of functors*,  $\rho_i: \mathcal{E}_i \Longrightarrow (\mathcal{F} \triangleright \mathcal{E}_{i+1})$ , where  $\mathcal{E}_i$  is the functor assigning schemas to the  $i$ -th fiber sketch. Only in the presence of the commutativity above the entire construction becomes a fibration in the technical categorical sense.

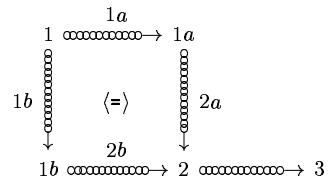
Thus, a constraint induced by the software reality leads to a mathematically justified construct. This remarkable fact may be considered as an instance of the general phenomenon: high relevance of the arrow language developed in category theory for software engineering (cf.[16, 15, 6]).  $\diamond$

**4.3** The construction of hypersketch fibration can be generalized in the following way. Till to now we assumed that the collection of hypersketches is organized in a refinement chain. However, it may happen that the schema repository is a set of such chains outgoing from a common source, for example, these chains can represent different directions of the design project (*eg*, directions  $a$  and  $b$  on Fig. 3). Moreover, there is nothing strange if these chains meet in some node (different projects lead to the same view hypersketch, *eg*,  $\mathcal{S}_2$  on Fig. 3) and then diverse again.

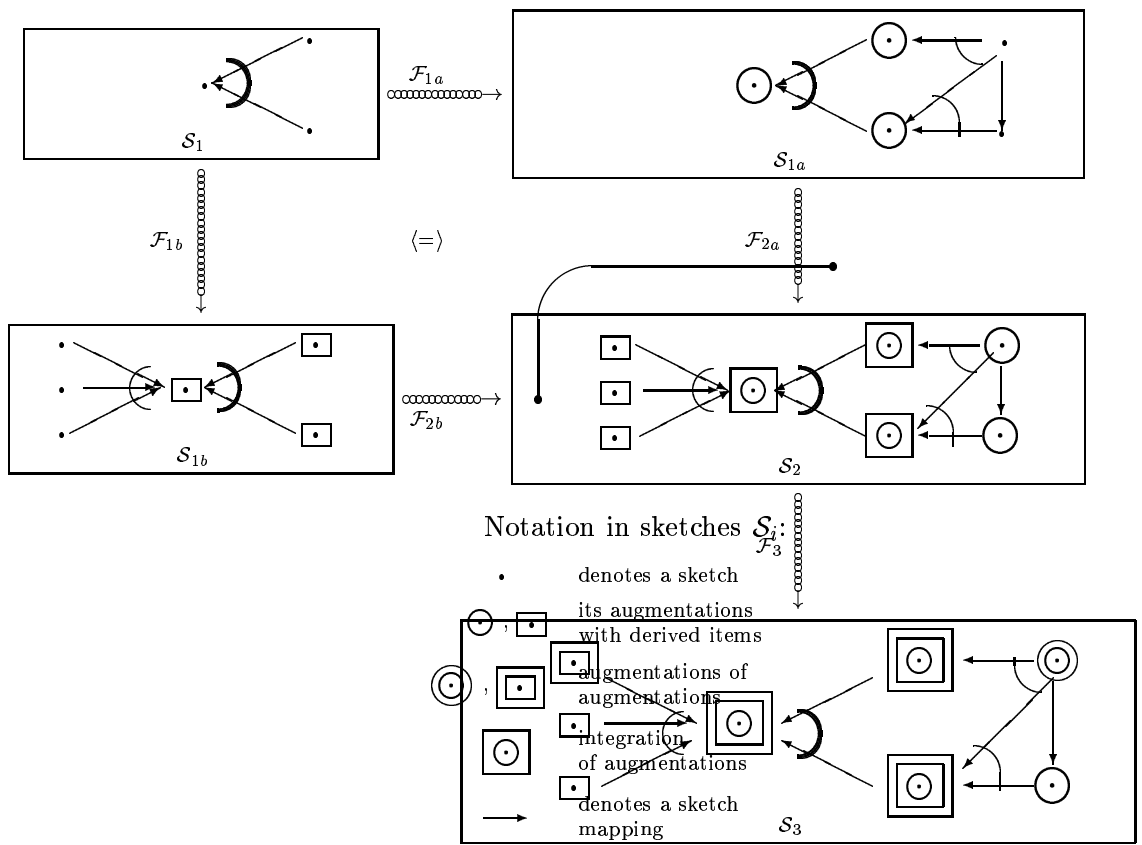
In other words, one can well assume that refinement steps (indexes) are organized into a graph  $I$  (the graph on Fig. 3(a) is an example) such that for each node  $i \in I$  there is a hypersketch  $\mathcal{S}_i$  (see Fig. 3b) coupled with some functor  $\mathcal{E}_i$  sending nodes of  $\mathcal{S}_i$  to schemas and arrows of  $\mathcal{S}_i$  to mappings (views) between schemas (these terminal schemas and mappings provided by  $\mathcal{E}_i$  are not shown on Fig. 3). Further, each arrow  $a: i \rightarrow j$  in  $I$  denotes a hypersketch refinement, that is, a sketch mapping  $\mathcal{F}_a: \mathcal{S}_i \rightarrow \mathcal{S}_j$  and a natural transformation (refinement as such)  $\rho_a: \mathcal{E}_i \Rightarrow \mathcal{F}_a \gg \mathcal{E}_j$  as it was explained above (in the example on Fig. 3, refinements  $\rho_a$ 's are not specified, their existence is only implicitly pointed out by circle and square frames over  $\bullet$ -nodes). In addition, the refinement steps can be subjected to some conditions, for example, two different refinement chains between common source and target may be required to be commutative. If one specifies these conditions as properties of some diagrams in the graph  $I$ , the latter will become a sketch, the *project hypersketch* (*eg*, the sketch  $\mathcal{P}$  on Fig. 3a).  $\diamond$

So, if the design process is properly documented, it will result in a vast multiple-level schema repository. As we have seen, the latter can be presented as a project hypersketch whose nodes are mapped to view hypersketches and arrows are mapped to their refinements. In its turn, nodes of view hypersketches are mapped to schemas stored in the library (which might be also sketches, or relational schemas, or other specifications of some predefined kind) and arrows of view hypersketches are mapped to view mappings also to be stored in the library. The entire construction may look monstrous but considering similar things is normal in category theory, and the corresponding notation, terminology and arrow machinery were developed. Thus, the view-refinement hypersketch structure gives a modularization mechanism for complex specifications, which, in contrast to other approaches to this infamous problem, is semantically and mathematically justified.

On the other hand, this structure is a proper generalization of the concept of schema grid proposed in the works of the Italian school [3]. Note, the schema grid concept was successfully tried in a series of projects performed for important institutions (amongst which are the Bank of Italy and the Italian State Administration). However, their grid is a very special case of our fibration when view hypersketches are discrete graphs without arrows. Actually, authors of [3] consider only the simple situation when views are subschemas and so each graph (fiber) of views can be replaced by a binary subschema relation on the view schemas. In effect, the construction of hypersketch fibration is a more precise explication of the intentions underlying their concept of schema grid.



a) Project sketch,  $\mathcal{P}$



b) Extension of  $\mathcal{P}$

Figure 3: Example of hypersketch fibration

## 5 Conclusion

Structuring specification repositories is an important issue of engineering business specifications. Languages suitable for this purpose should possess the following characteristics:

- be handy and transparent to the human in order to facilitate the design and operation;
- be precise and even formalizable to provide unambiguous understanding and computer implementation;
- be sufficiently expressive, at any rate, cover the concepts of view, schema integration, refinement;
- be polymorphic, that is, be based on patterns independent on specifics of languages used for component specifications – this feature is crucial for heterogeneous specification environment.

The task of designing such languages seems to be extremely difficult. Nevertheless, the arrow notation (and the formalism underlying it) described in the paper do provide a general framework where desired languages can be built. An essential advantage of the framework is that it is based on the firm ground of mathematical category theory. In fact, complex structures we have outlined are nothing but specialization of general constructions studied in category theory.

One more property of the framework is that it is algebraic in its nature. The algebra meant here is the algebra of diagram operations, that is, a *graphical algebra*. Correspondingly, the usual algebraic machinery of term rewriting is transformed into the so called *diagram chasing*, the technique which was polished in category theory to a great extent. Algebraicity of a notational machinery is an important advantage for adapting it for computer processing and automation.

Speaking in a wider context, a conjecture is that the arrow machinery (and the arrow style of thinking on a whole) are of extreme high suitability for managing complex specifications, especially in the heterogeneous environment. In particular, it was shown in the paper that works of the Italian school on structuring schema libraries can be naturally represented in the arrow framework which leads to a more general and unified setting. Another justification for the conjecture can be seen in the successful usage of the arrow framework in approaching the difficult task of heterogeneous database integration ([16]). One more example is the arrow approach to resolving structural conflicts in schema integration that was described in [4, 5]. These considerations might be summarized in the slogan: *the logic of structuring complex specification systems is the arrow logic*.

## References

- [1] S. Abiteboul and A. Bonner. Objects and views. In *Proc.ACM SIGMOD Conf. on Management of Data*, pages 238–247, 1991.

- [2] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall International Series in Computer Science, 1990.
- [3] C. Batini, G. Battista, and G. Santucci. Structuring primitives for a dictionary of entity relationship data schemas. *IEEE Trans.Soft.Engineering*, 19(4):344–365, 1993.
- [4] B. Cadish and Z. Diskin. Algebraic graph-based approach to management of multibase systems, I: Schema integration via sketches and equations. In *Next Generation of Information Technologies and Systems, NGITS'95*, 2nd Int.Workshop, pages 69–79, Naharia (Israel), 1995 (<ftp://ftp.cs.chalmers.se/pub/users/diskin/PAPERS-DB/ngits95.ps.gz>).
- [5] B. Cadish and Z. Diskin. Heterogenous view integration via sketches and equations. In *Foundations of Intelligent Systems*, Proc. 9th Int.Symposium, *ISMIS'96*, Springer LNAI'1079, pages 603–612, 1996.
- [6] Z. Diskin. Formalization of graphical schemas: General sketch-based logic vs. heuristic pictures. In *10th Int. Congress of Logic, Methodology and Philosophy of Science*, 1997, Kluwer Acad.Publ.(<ftp://ftp.cs.chalmers.se/pub/users/diskin/PAPERS-Math/lmpsci95.ps.gz>).
- [7] Z. Diskin. The arrow logic of visual modeling and taming heterogeneity of semantic models. In *Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, Technical Report, Munich University, 1998 (<ftp://ftp.cs.chalmers.se/pub/users/diskin/PAPER-DB/ecoop98/part1.ps>, [part2.ps](ftp://ftp.cs.chalmers.se/pub/users/diskin/PAPER-DB/ecoop98/part2.ps)).
- [8] Z. Diskin and B. Kadish. Variable set semantics for generalized sketches: Why ER is more object-oriented than OO. To appear in *Data and Knowledge Engineering*, manuscript is available by <ftp://ftp.cs.chalmers.se/pub/users/diskin/ER/ERvsOO.ps>.
- [9] Z. Diskin and B. Kadish. Variable sets and functions framework for conceptual modeling: Integrating ER and OO via sketches with dynamic markers. In *OOER'95: Object-Oriented and Entity-Relationship Modeling*, Proc. 14th Int.Conf., Springer LNCS'1021, pages 226–237, 1995.
- [10] Z. Diskin, B. Kadish, and F. Piessens. The Arrow Manifesto: Towards software engineering based on comprehensible yet rigorous graphical specifications. In *15th Int. Congress for Cybernetics*, 1998 (<ftp://ftp.cs.chalmers.se/pub/users/diskin/Manifest/namur98.doc>,[\\*.ps](ftp://ftp.cs.chalmers.se/pub/users/diskin/Manifest/namur98.ps)).
- [11] C. Francalanci and B. Pernici. Abstraction levels for entity-relationship schemas. In *13th Int.Conf. ER'94*, Springer LNCS'881, pages 456–473, 1994.
- [12] J. Goguen. Semiotic morphisms. Technical report, University of California at San Diego, 1997. TR-CS97-553.
- [13] J.A. Goguen. A categorical manifesto. *Mathematical structures in computer science*, 1(1):49–67, 1991.
- [14] S.J. Hegner. Pairwise-definable subdirect decomposition of general database schemata. In *3rd Symp.MFDBS'91*, Springer LNCS'495, pages 243–257, 1991.
- [15] M. Johnson and C.N.G. Dampney. On the value of commutative diagrams in information modeling. In *Algebraic Methodology and Software Technology, AMAST'93*. Springer, 1993.

- [16] L. Kalinichenko. Methods and tools for equivalent data model mapping construction. In *Advances in Database Technology - EDBT'90*, pages 92–119, 1990.
- [17] I.S. Mumick. The rejuvenation of materialized views. In *6th Int. Conf. CISMOT'95*, Springer LNCS'1006, pages 258–264, 1995.
- [18] G. Santucci, C. Batini, and G. Battista. Multilevel schema integration. In *12th Int. Conf. ER'93*, Springer LNCS'823, pages 327–338, 1993.
- [19] M. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In *2nd Int. Conf. DOOD'91*, Springer LNCS'566, pages 189–207, 1991.
- [20] M.H. Scholl, H.-J. Schek, and M. Tresch. Object algebra and views for object bases. In *Int. Workshop on Distributed Object Management, Edmonton, Canada*, 1992.
- [21] C. Tuijn and M. Gyssens. Views and decompositions from a categorical perspective. In *4th Int. Conf. on Database Theory, ICOT'92*, Springer LNCS'646, pages 99–112, 1992.

## A Data modeling via sketches

### A.1 Semantic modeling via sketches

The sketch data model was introduced in [9], a detailed description can be found in [7]. Syntactically, a sketch is a directed multigraph in which some diagrams are labeled with special markers. The intended semantics is to interpret nodes as sets and arrows as functions (both evolving in time) while a marker is interpreted as a certain (constant) constraint imposed on the diagram of sets and functions labeled by the marker. In other words, nodes denote classes and value domains, arrows denote references and attributes, and markers denote integrity-checking procedures (a collection of diagram markers is presented in Table 1). In this way one obtains a variety of sketch data models by choosing one or another signature of markers. In was outlined in [7] how various ER- and OO-oriented data model can be simulated by sketches in appropriate signatures. On the other hand, sketches in different signatures can be readily considered similar in the common joint signature.

Let us consider an example of semantic modeling via sketches. The universe we wish to model is as follows. Suppose, the user is interested in information about men, married couples and married women of some town for, say, the last 50 years. Here "married women" means women which are or have been married during the last 50 years. A rough view on the universe is described by the ER-diagram on the top of Fig. 4 in a conventional ER-notation. Semantics of nodes and attributes is hopefully clear from their names, *MDate*, *DDate* are dates of marriage and divorce (the latter is optional). *BDate* is a tuple type determined by its own attributes *Day:Int*, *Month:Int*, *Year:Int*. The domain of the optional attribute *Character* is a set consisting of three values *a, c, q*. The class *Woman* is of weak entity type since users are assumed to be interested only in women which are or have been married.

Note, the ER-diagram does not reflect one essential point in the semantics of *Married*-objects: the diamond means that the pair (*wife,husb*) is a key whereas, in general, this

Semantic situation:  
 users are interested  
 in information  
 (say, for the last 50 years)  
 about men, married  
 couples and married women.

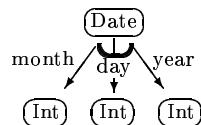
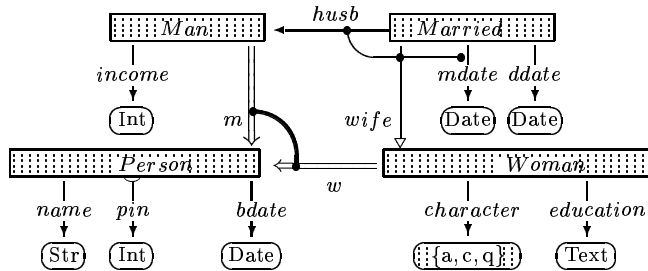
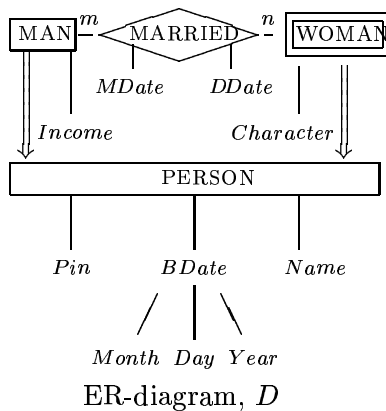


Figure 4: Semantic modeling via sketches

Table 1: A collection of diagram predicates (constraints)

Name	Arity Shape and Designation	Denotational Semantics
Separating Source	$  \begin{array}{c}  X \\  \begin{array}{ccc}  f_1 & \dots & f_n \\  \swarrow & \dots & \searrow \\  Y_1 & \dots & Y_n  \end{array}  \end{array}  $	$(\forall x, x' \in X) x \neq x' \text{ implies } (\exists i) f_i(x) \neq f_i(x')$
Monic Arrow <sup>†</sup>	$  \begin{array}{c}  X \xrightarrow{f} Y  \end{array}  $	$(\forall x, x' \in X) x \neq x' \text{ implies } f(x) \neq f(x')$
ISA-Arrow	$  X \xrightarrow{f} Y \quad \text{or} \quad X \subset \xrightarrow{f} Y  $	$X \subset Y \text{ and } f(x) = x \text{ for all } x \in X$
Covering Flow	$  \begin{array}{c}  Y \\  \begin{array}{ccc}  f_1 & \dots & f_n \\  \swarrow & \dots & \searrow \\  X_1 & \dots & X_n  \end{array}  \end{array}  $	$(\forall y \in Y)(\exists i < n) y \in f_i(X_i)$
Cover <sup>†</sup>	$  \begin{array}{c}  X \xrightarrow{f} Y  \end{array}  $	$Y = f(X)$
Disjoint Images	$  \begin{array}{c}  Y \\  \begin{array}{ccc}  f_1 & \dots & f_2 \\  \swarrow & \dots & \searrow \\  X_1 & & X_2  \end{array}  \end{array}  $	$f_1(x_1) \cap f_2(x_2) = \emptyset$
Disjoint Covering Flow	$  \begin{array}{c}  Y \\  \begin{array}{ccc}  f_1 & \dots & f_n \\  \swarrow & \dots & \searrow \\  X_1 & \dots & X_n  \end{array}  \end{array}  $	$(\forall y \in Y \exists i < n) y \in f_i(X_i)$ and $i \neq j \text{ implies } f_i(X_i) \cap f_j(X_j) = \emptyset$

is not the case. Indeed, it is well possible that a married couple got divorced and then got married again. Thus, the correct identifier is the triple  $(wife,husb,mdate)$ . Diagram is drawn as shown since in the notational ER-standard there are no graphical means to express the required semantics. In contrast, this can be easily done with the sketch machinery: the sketch specifying the situation is depicted on the lower figure (see Table 1 for the meaning of the marked diagrams). Note also monic markers on the key attribute *pin* and the cover marker on the arrow *wife*.

Rectangle nodes are *abstract classes* whose extensions should be stored in the DB: this is additionally pointed by small dots filling-in rectangles. Oval nodes are predefined *value domains* whose semantics is *a priori* known to the DBMS. For the sketch approach,  $\text{Int}$  and  $\{a, c, q\}$  are *markers* (in our precise sense) hung on corresponding nodes, that is, constraints imposed on their intended semantic interpretations. For example, if a node is marked by  $\text{Int}$  its intended semantics is the predefined set of integers.

It seems that distinguishing node names and node markers in semantic schemas is a precise formal description of a well known differentiation between abstract and printable classes.

## A.2 Derived information via diagram operations

Consider, again, the simple universe described on Fig. 4. An important constraint that should be added to the schema is the condition of unique identification of any currently married couple by either its husband, or its wife as well. In other words, the subset of the

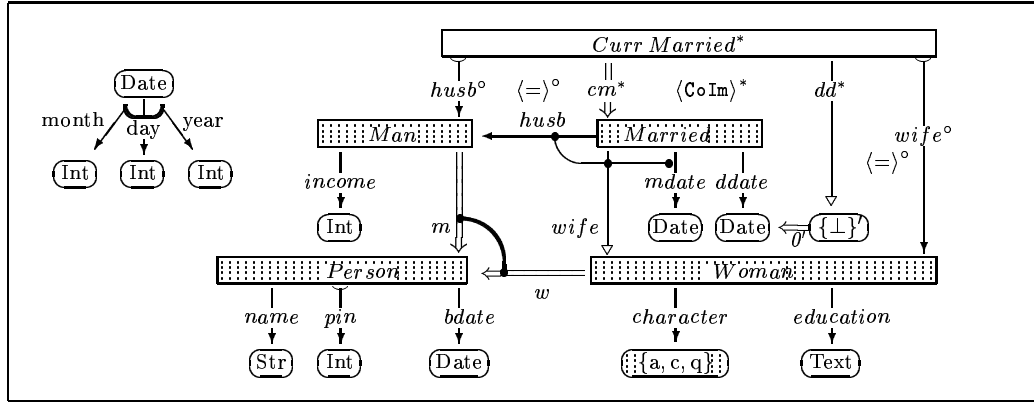


Figure 5: Sketches with operation markers: Specifying a query to a sketch = Extending it with derived items

relation ‘Married’ for which the attribute ‘*ddate*’ is undefined, is of the one-one relationship type. To express such a constraint, one should extend the original sketch with derived items as shown on Fig. 5: in order to distinguish basic items from the derived ones, the former are filled-in with small dots intended to remind about the extension to be stored. Of course, derived arrows should be also specially distinguished, and we agree to denote derived items by hanging various superscripts on their names (like ‘ $\circ$ ’, ‘ $*$ ’, ‘*etc.*’).

Certainly, the derived items of the sketch we consider can be obtained by an evident **Select-From-Where** SQL-query. Equally, this query can be specified as a composition of elementary diagram operations with sets and functions: *Null*, *CoImage*, *Composition*, presented in the top of Table 2 (as for the *Null* operation, we assume that each domain contains a single distinguished null value). Denotational semantics of operations is described in the corresponding column of the table.

A diagram operation  $F$  is specified by a sketch denoting its input data,  $S_F^{\text{in}}$ , and a sketch denoting the output data,  $S_F^{\text{out}}$ . It is convenient, however, to join  $S_F^{\text{in}}$  and  $S_F^{\text{out}}$  into one sketch  $S_F$  so that an operation  $F$  is specified by an inclusion  $\iota_F : S_F^{\text{in}} \hookrightarrow S_F$  (in the Table 2 sketches  $S_F$ ’s are called output sketches). The body of operation is then a procedure  $\llbracket F \rrbracket$  which calculates an extension of  $S_F$  from a given extension of  $S_F^{\text{in}}$ .

In some conventional linear notation, the derived items of the sketch  $\bar{S}$  on Fig. 5 can be presented as follows:

**define** ( $0'$ ,  $\{\perp'\}$ ) **as** *Null* (*Date*)  
**define** (*CurrMarried\**,  $cm^*$ ,  $dd^*$ ) **as** *CoImage* ( $\{\perp'\}'$ , *ddate*,  $0'$ )  
**define** ( $husb^\circ$ ) **as** *Composition*( $cm^*$ , *husb*)  
**define** ( $wife^\circ$ ) **as** *Composition*( $cm^*$ , *wife*)

A disadvantage of this notation is that connections between items above are hidden. In contrast, in the graphical sketch framework they are seen immediately.

It is important to distinguish the origin of the ISA-marker that labels the arrow  $cm^*$  from that of the monic markers labeling arrows  $husb^\circ$  and  $wife^\circ$ . The ISA-marker on  $cm^*$  is a postcondition of the operation *CoImage*: the corresponding constraint should be satisfied automatically provided the body of the procedure satisfies the specification.

Table 2: A collection of diagram operations (queries)

Name (marker)	Arity Shape		Denotational Semantics	Linear notation
	Input sketch	Output sketch		
Null	$\bullet A$	$A \xleftarrow[0]{\perp} \{\perp\}$	$0(\perp) = \perp$	
Coimage (CoIm)	$X \xrightarrow{f} Y$ $B$ $b \Downarrow$	$B' \xrightarrow{f'} B$ $b' \Downarrow$ $X \xrightarrow{f} Y$ $b \Downarrow$	$B' = \{x \in X : fx \in B\}$ $f' = \text{restriction of } f \text{ on } B'$	$B' = f^{-1}(Y)$ or else $B' = \text{CoIm}_f(B)$
Image (Img)	$X \xrightarrow{f} Y$	$I \xrightarrow{i} Y$ $f' \uparrow$ $X \nearrow f$	$I = \{f(x) : x \in X\}$ $(\forall x \in X) f'(x) = f(x)$	$I = f(X)$ , or else $I = \text{Im}_f(X)$
Composi- tion (=)	$Z \xrightarrow{g_2} Y$ $g_1 \uparrow$ $X$	$Z \xrightarrow{g_2} Y$ $g_1 \uparrow$ $X \nearrow f$	$(\forall x \in X) f(x) = g_2(g_1(x))$	$f = g_1 \blacktriangleright g_2$
Graph (Gra)	$X \xrightarrow{f} Y$	$G$ $p \swarrow \curvearrowright \searrow q$ $X \xrightarrow{f} Y$	$G = \{(x, fx) : x \in X\}$ $p, q$ are projections	$G = \text{Gra}(f)$
Difference (Dif)	$A$ $\Downarrow$ $X$	$A$ $\Downarrow$ $X \xleftarrow{\perp} A'$	$A' = \{x \in X : x \notin A\}$	$A' = X \setminus A$

There are similar automatically satisfied postcondition markers in specifications of other operations: such a marker is present in the output sketch of operation (see Table 2). In contrast, monic markers (denoting one-one functions) hung on the arrows *wife*<sup>o</sup>, *husb*<sup>o</sup> express the actual constraint arising from semantics of the schema, *a priori* these arrows should not be monic.

*Remark on notation.* It would be convenient to depict automatically satisfied markers, as well as derived items and operation markers by one colour, say, green, whereas markers denoting actual constraints by another colour, say, red. Our practice shows that colored sketches are quite readable despite a large number of conventional signs.

Thus, in the sketch  $\bar{S}$  on Fig. 5, nodes *CurrMarried* and  $\{\perp\}'$  as well as arrows *husb*<sup>o</sup>, *cm*<sup>\*</sup>, *dd*<sup>\*</sup>, *wife*<sup>o</sup>, *0'* are green whereas small tail arcs on *husb*<sup>o</sup>, *wife*<sup>o</sup> are red. Two markers (=) and the marker CoIm are green (since they are operation markers) while the arc spanning the arrows *husb*, *wife*, *mdate*, the arc spanning *m*, *w*, the head of the arrow *wife* and the tail of *pin* are red. Other items are black. To make the sketch more understandable the reader is encouraged to paint it!

### A.3 Composition of diagram operations (queries)

Queries can be composed by passing a part of the output data of a query (or a set of queries) to the input of another query operation. The way of passing is specified by the corresponding mappings of sketches. On the other hand, the composition can be presented as a stepwise augmentation of the initial schema with derived items. In fact, we have already used this procedure in the process of building the sketch on Fig. 5. A

bit more involved example is presented on Fig. 1(a) in the main text where we omit the fragment specifying derivation of items *CurrMarried*, *cm\** described in Fig. 5.

Given Table 2 with conventions adopted above, the graphical image on Fig. 1(a) specifies a system of queries against the basic sketch in a unambiguous way. For example, the marker  $\langle \text{CoIm} \rangle^{\S}$  labeling the the square diagram around it specifies that the extent of the node  $Man^{\S}$  consists of those objects of the class *Man* for which the value of the attribute *income* is greater than  $10^6$ . Similarly, the extension of the node  $Man^!$  is the intersection of extensions of  $Man^{\S}$  and  $Man'$  (note, the *CoImage* operation applied to two ISA-arrows turns into the ordinary intersection of sets). The node  $Man'$  is produced by the operation *Image* applied to the arrow *husb*<sup>o</sup> which is derived, in its turn, by the composed query presented on Fig. 5. It follows from the semantics of these procedures that, *eg*, the extension of the class  $Man^!$  consists of those happy *Man*-objects which are married (at the present time) and rich (with *income* no less than  $10^6$ ). Certainly, such a query can be expressed by standard SQL-means but the discussion of which of the languages is better is not relevant for our present considerations. All that we wish to demonstrate in this respect is how standard SQL-queries can be expressed by diagram operations over sketches.

To summarize, it should be stressed that what was proposed is not a single query language but rather a framework for constructing graph-based query languages. Everyone can choose that signature of operations one prefers, and what was suggested is how to specify graph-based operations in a proper way independently on any syntactic details (a correct framework can be easily sugared afterwards). On the other hand, the proposed framework is really universal: it is proven in category theory that any query language possessing a formal semantics can be simulated by an appropriate signature of diagram operations over sketches.