Universal Arrow Foundations for Visual Modeling

In *Diagrams'2000*: Proc. 1st Int. Conference on the theory and application of diagrams, Edinburgh (Scotland), UK, Sept. 1-3, 2000. M. Anderson, P. Cheng, V. Haarslev (Eds.), Springer LNAI#1889, 2000, pp.345-360

Zinovy Diskin[1], Boris Kadish[1], Frank Piessens[2] and Michael Johnson[3]

[1] Lab for Database Design, *Frame Inform Systems*, Ltd., Riga, Latvia (zdiskin@acm.org)
Supported by Grants 93.315 and 96.0316 from the Latvia Council of Science
[2] Postdoctoral Fellow of the Belgian National Fund for Scientific Research, Dept. of Computer Science, K. U. Leuven, Belgium

[3] School of Mathematics and Computing, Macquarie University, Sydney, Australia

Abstract. The goal of the paper is to explicate some common formal logic underlying various notational systems used in visual modeling. The idea is to treat the notational diversity as the diversity of visualizations of the same basic specificational format. It is argued that the task can be well approached in the arrow-diagram logic framework where specifications are directed graphs carrying a structure of diagram predicates and operations.

1 Introduction

Compact graphical images, diagrams, are very often nothing but *visual models* presenting various aspects of the universe of discourse in a comprehensible and easy to communicate way. Proper modeling is a key to a proper design and indeed, diagrams have proved their practical helpfulness in a wide range of design activities: from thinking out how to erect a shack (where a drawing is useful yet optional) to the design of high-rise buildings of business/software systems (impossible without diagrams). In general, the history of graphical notations invented in various scientific and engineering disciplines is rich and instructive but it is software engineering (SE) where during recent years one can observe a Babylonian diversity of visual modeling languages and methods: entity-relationship (ER) diagrams and a lot of their dialects, object-oriented (OO) analysis schemas in a million of versions and, at last, the recent Unified Modeling Language (UML) which itself comprises a host of various notations.

An important peculiarity of diagrams used in SE is the intention to provide them with very precise, or even formal, meaning. That is, the meaning $\mathcal{M}(D)$ of a diagram D is considered to be described in precise (ideally mathematical) terms. This latter description forms some precise specification S_D possessing precise (formal) semantics $M(S_D)$. Thus, $M(S_D)$ is a formal abstraction of the intuitive meaning $\mathcal{M}(D)$ and we consider S_D as some *(internal) logical specification hidden in* D. It seems the pattern just described is typical (or, at least, is desired to be typical) for diagram usage in SE.

The diversity of domains where diagrams are used is vast (even inside SE), correspondingly, we have a vast diversity of intuitive universes from which meanings $\mathcal{M}$ have to be taken. However, as soon as we begin to speak about their formal reducts M, the diversity is compressed into only a few universes of mathematical constructs. Each of the latter can be described in its own specific language, say, the language of set theory, or type theory, or higher order logic, or categorical logic.[1] In fact, it is proven that all these languages are of equal, and universal, expressive power so that any formal semantics can be expressed in any of them. So, S_D above can be taken to be a specification (theory) in any of the formal languages mentioned. However, one of them is of special interest for us because specifications in this language also appear in a diagrammatic graphic form. We mean categorical logic where specifications are nothing but graphs of nodes and arrows, in which some fragments are marked by labels denoting predicates taken from a predefined signature; this graphical format is called a *sketch*. In fact, we always deal with Π-sketches where Π denotes a signature of predicate labels (markers).

So, if the formal model for the meaning of D is also specified in a graphical language $\mathbf{Ske}(\Pi)$, then one can hope for some useful correspondence, even some visual similarity, between the external visual appearance D and the internal logical specification, sketch S_D, behind it. In this framework, the diagram D appears as a visualization, most often, a special abbreviation, of the precise and detailed logical specification S_D. For example, the language of ER-diagrams determines a sketch signature Π_{ER} such that any ER-diagram D can be presented as a special visualization of the corresponding Π_{ER}-sketch S_D (section 3).

This gives rise to a general thesis that *any diagram with precise semantics (to be described in mathematical terms) actually hides a sketch in a suitable signature of markers*. Then, any diagrammatic notation (language) $\mathcal{L}$ appears as a special visualization superstructure $Vis_{\mathcal{L}}$ over a certain basic specification sketch language $\mathbf{Ske}(\Pi_{\mathcal{L}})$ where $(\Pi_{\mathcal{L}})$ is a signature of markers corresponding to $\mathcal{L}$. This means that any $\mathcal{L}$-specification, a diagram D, can be presented as $D = Vis_{\mathcal{L}}(S_D)$ with S_D a $\Pi_{\mathcal{L}}$-sketch specifying the meaning of D in precise terms and $Vis_{\mathcal{L}}$ a mapping sending $\Pi_{\mathcal{L}}$-sketches into their visual presentations. In this way the *diversity* of diagrammatic notations can be transformed into a *variety* of sketch models in different signatures. Indeed, sketches in different signatures are nevertheless sketches, and they can be uniformly compared via relating/integrating their signatures. Though the latter task is far from being trivial, it is precisely formulated and can be approached by methods developed in category theory.

The view outlined above opens wide and tempting possibilities for the uniform treatment of many diagrammatic notations and for putting them on precise mathematical foundations, refining their vocabularies and providing them with formal semantics. It brings much more discipline to the art of designing new diagrammatic notations and provides a base for the systematic treatment of the inter-language

[1] *Categorical logic* is a discipline of viewing and studying logic within the *category theory* (CT) framework. The latter is a branch of modern algebra where mathematical structures are considered in a graph-based specification methodology; a standard reference suitable for computer science is [4].

issues: translation and integration of diagrams in different graphical languages. In addition, it opens the door for incorporating powerful algebraic techniques developed in CT into the field of visual modeling. In particular, diagram transformation is reduced to a graph-based counterpart of algebraic term rewriting – the so called *diagram chasing* (see [7] for an example of useful application) elaborated in CT to a great extent.

Of course, the realization of this idea is a wide research program: the sketch treatment of each particular diagrammatic notation needs careful elaboration. Indeed, a diagram language really used in a scientific/engineering domain accumulates a lot of useful "notational tips", habits and traditions. A correct sketch approach to such a language means putting it on precise semantic foundations and then explaining/specifying the notational peculiarities as visualizations of the underlying logical constructs rather than removing them entirely as "non-logical". Of course, while the sketch language as such is a mathematical phenomenon, visualization of sketches in one or another diagrammatic notation easy to use and communicate is a highly non-trivial issue of cognitive, and even more generally, cultural nature far beyond formal mathematics and logics. Thus, the research program in question is essentially interdisciplinary.

The plan of the paper is as follows. Section 2 presents general foundations of the sketch specification machinery. In section 3 several concrete applications in the so called semantic (conceptual) data modeling are considered: we demonstrate the sketch treatment of classic ER-diagrams and the recently fashioned UML. Though these applications are taken from a particular field, they illustrate an approach of (we emphasize once again) quite general nature and applicability.

2 Basics of arrow thinking and arrow logic

It is important for understanding the purposes of the present paper to recognize the distinction between string-based and graph-based logics ([3, 10, 12]). The point is that any specification – as it is presented to its reader – is actually a *visual presentation* of a certain underlying *logical specification* as such. In general, there are possible linear string visualizations of graph-based specifications and, conversely, graphical visualizations of string-based logical specifications. Examples of the latter are the well-known Euler-Venn diagrams for visual presentation of propositional logic statements, or the graphic representation of first-order logic sentences by conceptual graphs [16] or the visual presentation of first-order logic used in Barwise and Etchemendy's *Hyperproof* [5], or even the graphical interfaces to relational database schemas used in many design tools. Conversely, graph-based logic specifications can be presented in a linear plain form: after all, a graph is an ordinary two-sorted mathematical structure well specified by formulas.

So, in considering graphic notational systems one should carefully distinguish between specification and visualization, and graph-based logics should be carefully distinguished from graphical interfaces to string-based logics. Any logic where arities of predicate/operation symbols are strings is a string-based logic, no matter how graphical its visualization looks.

2.1 Arrow diagram logic: simple examples

By definition, one deals with a *graph-based logic* if arity shapes of predicates are graphs (of place-holders). In other words, a predicate arity consists of node place-holders and arrow place-holders organized into a directed graph.[2] A typical case when such a logic naturally arises is when one deals with specifying systems of sets and functions as shown in Table 1.

The predicate of *set inclusion* (the 1st row) is actually a property of a diagram consisting of two nodes (sets) A, B and an arrow (mapping) between them (2nd column). If this property is declared for a diagram of the shape just described (3rd column) – this is visualized by the block-body figure of the arrow – it means that the source set is a subset of the target and the mapping is their inclusion.

A collection of subsets may be disjoint; this property is specified by an arrow diagram predicate as shown in the 2nd row of the table.

A collection of mappings into a common target set may have the *covering* property: each element in the target set is a value of at least one of the mappings. This predicate is expressed as shown in 3rd row.

The predicate of *separating* arrow family (the 4rd row) is somewhat dual to the covering predicate (in category theory, this duality can be expressed in precise terms). It can be declared for a diagram consisting of a source node and a family of arrows going out of it as it is shown in the 2nd column. The formal meaning is as follows:

(1-1) for any $x, x' \in X,\ x \neq x'$ implies $f_i(x) \neq f_i(x')$ for some i

Note however that in such a case the tuple-function $f = \langle f_1 \dots f_n \rangle$ into the Cartesian product of domains,

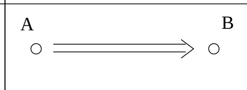
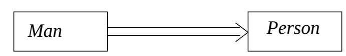
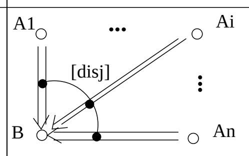
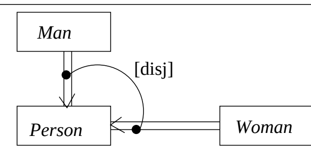
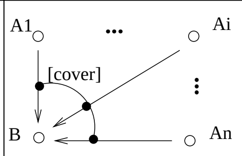
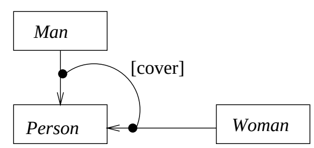
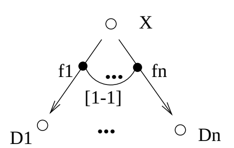
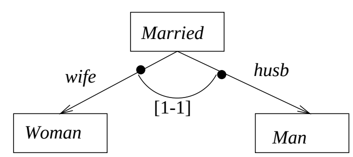
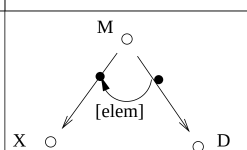
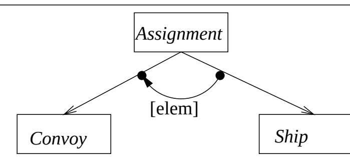
$$f = \langle f_1 \dots f_n \rangle \colon\ X \longrightarrow D_1 \times \cdots \times D_n, \quad fx \stackrel{\text{def}}{=} \langle f_1 x, \dots, f_n x \rangle$$

is injective (one-one) so that elements of X can be considered as unique names for tuples from a certain subset of $D_1 \times \ldots \times D_n$, namely, the image of f. In fact, elements of X can be identified with these tuples so that X is a relation up to isomorphism. In the classical ER-terminology [8], if the domains D_i's are *entity* sets then f_i's are *roles* and any $x \in X$ is a *relationship* between entities $f_1(x), \dots, f_n(x)$. Then, for example, the right-most diagram in the row means that *Married*-objects are nothing but pairs $(wife, husb)$ whose components are taken from classes *Woman*, *Man* respectively. Thus, we have precisely expressed the internal tuple-structure of X-objects by declaring the corresponding property for some arrow diagram adjoint to the node X.

We may proceed in a similar way for specifying other types of structures. For example, the intended meaning of the Grouping (SubPowerset) predicate (5th row) is to state that the set X consists of subsets of the set D, in other words, of groups

[2] In general, there are richer graph-based shapes, *eg*, *2-graphs* with 2-arrows between arrows, *3-graphs* with 3-arrows between 2-arrows and so on.

Table 1. Sets-and-functions predicates in a graph-based notation

Predicate Name	Arity shape with visualization	Example of declaration on a diagram
Set Inclusion		
Disjointness		
Covering		
Separating family of functions		
Grouping (SubPowerset)		

of D-objects (e.g., objects of class *Convoy* are groups of *Ship*-objects). Again, in the arrow diagram logic this structure of the X-elements is expressed by declaring a certain predicate (denoted by, say, [elem]) for a certain diagram adjoint to the node X. We omit the technical description.

2.2 Arrow diagram logic: general formulation

Examples considered above may be summarized as follows. Any given diagram property has a predefined shape, that is, a configuration of nodes and arrows for which the property makes sense. So, in the graph-based logic a predicate is specified by its name and a graph (of place-holders) called the *(logical) arity shape* of the predicate (see the middle column of Table 1). In addition, the arity shape should be supplied with auxiliary graphic means like arcs or double-body arrows for visualizing predicate declarations on schemas. Of course, these auxiliary means do not occur in the logical arity as such – together with the latter they form the *visual arity shape* providing the visualization mechanism.

In order to declare a predicate P with some arity shape G_P for a system $\mathcal{S}$ of sets and functions, one must assign $\mathcal{S}$-sets to nodes in G_P and $\mathcal{S}$-functions to arrows in G_P in such a way that adjointness conditions between nodes and arrows are respected. This can be considered as (i) arranging (names of) items from $\mathcal{S}$ into a graph so that the system $\mathcal{S}$ can be identified with this graph, and (ii) setting up a graph mapping $d\colon G_P \to \mathcal{S}$ from the arity graph into the graph of names. Such a mapping can be visualized by labeling nodes and arrows in the shape by names (of, respectively, nodes and arrows in $\mathcal{S}$, see the rightmost column in Table 1). The couple (predicate_marker, its shape labeled by names), *ie*, (P, d) in our notation, is a predicate declaration for the collection of items whose names were used as labels. Such a pair is called a *marked diagram* and so, in the graph-based logic, a predicate declaration is nothing but a marked diagram.[3] Thus, a graph-based logical specification is a graph of names, in which some diagrams are labeled with markers from some predefined signature. As it was said, we call such specifications *sketches* or, more precisely, Π-sketches where Π is the name of the signature.

An important peculiarity of the machinery is a hierarchical structure of sketch signatures. The point is that the arity shape of a predicate P may itself contain predicate markers defined prior to P, e.g., the disjoint predicate in 2nd row of the table uses inclusion. In such a case a P-predicate declaration in a schema, that is, a P-diagrams $d\colon G_P \to \mathcal{S}$ must fit these markers in G_P. For example, if an arrow f in G_P is marked as inclusion, its image, function $d(f)$ in $\mathcal{S}$, must be also an inclusion. Thus, in general, G_P is a sketch rather than graph and d is a sketch mapping, that is, a graph mapping compatible with markers.

The last two examples in Table 1 demonstrate a key feature of the arrow logic: nodes are always considered homogeneous and their internal structure (the structure

[3] So, in the sketch framework, the term 'diagram' is used in some narrow technical sense as above. In contrast, in engineering disciplines the term 'diagram' often refers to a graphical specification considered as a whole; to avoid misunderstanding, in the latter case we will use the term 'schema'.

of their elements) is specified by declaring the corresponding property of the corresponding arrow diagram adjoint to the node. Actually, this gives rise to a special way of thinking out specifications, the so called *arrow thinking*, and in the next section we will briefly manifest the general foundations of the approach.

2.3 Arrow thinking and category theory: a brief manifesto

Category theory (CT) is a modern branch of abstract algebra. It was invented in the forties and since that time has achieved a great success in providing a uniform structural framework for different branches of mathematics, including metamathematics and logic, and stating foundations of mathematics as a whole as well. CT should be also of great interest for scientific and engineering disciplines, in particular, software engineering, since it offers a general methodology and machinery for specifying complex structures of very different kinds. In a wider context, the 20th century has been the age of structural patterns, *structuralism*, as opposed to *evolutionism* of the previous century, and CT is a precise mathematical response to the structural request of our time.

The basic idea underlying the approach consists in specifying any universe of discourse as a collection of *objects* and their *morphisms* which normally are, in function of context, mappings, or references, or transformations or the like between objects. As a result, the universe is specified by a directed graph whose nodes are objects and arrows are morphisms. Objects have no internal structure: everything one wishes to say about them one has to say in terms of arrows. This feature of CT can be formulated in the OO programming terms as that object structure and behaviour are encapsulated and accessible only through the arrow interface. Thus, objects in the CT sense and objects in the OO sense have much in common.

A surprising result discovered in CT is that the arrow specification language is absolutely expressible: any construction having a formal semantic meaning[4] can be described in the arrow language as well. Moreover, the arrow language is proven to be a powerful conceptual means: if basic objects of interest are described by arrows then normally it turns out that many derived objects of interest can be also described by arrows in a quite natural way. The main lesson of CT is thus that to define properly the universe we are going to deal with it is necessary and sufficient to define morphisms (mappings) between objects of the universe. In other words, as it was formulated by a founder of categorical logic Bill Lawvere, *to objectify means to mappify.*

The arrow language is extremely flexible. In function of context, objects and arrows can be interpreted by: sets and functions (in *data modeling*), object classes and references (*OO analysis and design*), data types and procedures (*functional programing*), propositions and proofs (*logic/logic programing*), interfaces and processes (*process modeling*), states and transactions (*transaction modeling*), specifications and their mappings (*meta-modeling*). Moreover, all this semantic diversity is managed within the same syntactical framework of *sketch* specifications.

[4] that is, expressible in a formal set theory

3 Example of sketching a visual modeling discipline: Semantic data modeling via sketches

Often, within the same field of applying visual modeling (VM) there are many different VM-methodologies that have resulted in different notational systems. For example, in semantic (conceptual) data modeling there are different ER-based notations, and a lot of OO-based, and many "this-vendor-tool-based" ones. In fact, many specialists in semantic modeling and DB design use their own graphic specification languages which they find more suitable and convenient for their purposes. This is natural and reasonable but with the current trend to cooperative/federal information systems, one necessarily encounters severe problems with integrating specifications describing the same universe (or its overlapping fragments) in different notational systems. Several attempts to manage the semantic models heterogeneity were made both in academia (*eg*, [2]) and industry where the famous UML was recently adopted as a standard in visual modeling and design. However, while UML is indeed a significant industrial achievement towards unification, from the logical view point it is just another (monstrous) notational system rather than a framework to manage the heterogeneity problem.

3.1 General methodology

The main idea underlying the sketch approach to data modeling is to consider object classes as plain sets consisting of internally unstructured (homogeneous) elements whereas all the information about their type is moved into certain arrow (reference) structures adjoint to classes. Examples of how it can be done were demonstrated in section 2.2. Associations (relationships) between classes are also treated as sets (of tuples) with outgoing projection mappings and thus a semantic data schema in any notation can be considered as a specification of a collection of set-like and mapping-like objects subjected to certain constraints. A natural way to structure such data is to organize these objects into a graph, the *underlying graph* of the specification, and to convert constraints into predicates declared for the corresponding diagrams in the underlying graph. Before such structuring can be performed, the collection of diagram predicates which may be declared for set-and-mappings diagrams is organized into a *signature*, say, Π, and then a semantic specification is converted into a structure which we have called Π*-sketch*: it is a graph in which some diagrams are marked by predicate symbols from Π. In this way the diversity of semantic data models can be considered as the diversity of visualizations over the same basic sketch format and thus we have a particular case of the general approach described in the introduction.

3.2 Interpretation of arrows

The key point in the semantics of sketches is how to interpret arrows. In the former considerations we interpreted arrows by ordinary functions, *ie*, totally defined single-valued functions. This is the standard category theory setting. In contrast, in semantic modeling it is convenient (and common) to use optional and multivalued attributes/references, and so two additional different interpretations of arrows arise: by partially defined functions (p-functions) and by multivalued functions (m-functions). Their accurate sketch treatment needs a special explanation.

M-functions and p-functions are not ordinary functions subjected to some special constraints. Just the opposite, a single-valued function is a special m-function $f : A \twoheadrightarrow B$ when for any $a \in A$ the set $f(a) \subset B$ consists of a single element. Similarly, a totally defined function is a special p-function $f : A \circ\!\!\longrightarrow B$ whose domain $D_f \subset A$ coincides with the entire source set A. So, to manage optional multi-valued attributes and references in the sketch framework we assume that

(i) all arrows are by default interpreted by pm-functions,

(ii) there is an arrow predicate (marker) of being a single-valued function,

(iii) there is an arrow predicate (marker) of being a totally defined function.

It is convenient to visualize constraintless arrows (without markers) by $\circ\!\!\twoheadrightarrow$ whereas $\twoheadrightarrow$ and $\circ\!\!\longrightarrow$ are denotations of arrows on which markers are hung: the ordinary tail is the marker of being totally defined and the ordinary head is the marker of being single-valued. Of course, superposition of these markers is also legitimate and it is natural to visualize it by the arrow $\longrightarrow$. Thus, visualization of predicate superposition equals superposition of visualizations: here we have a simple instance of applying a useful general principle that reasonable graphic notation should follow.

3.3 Sketches vs. ER-diagrams: simple example.

The semantic situation we wish to model is as follows. Suppose, the user is interested in information about men, married couples and married women of some community for some time period; here "married women" means women who are or have been married during the period.

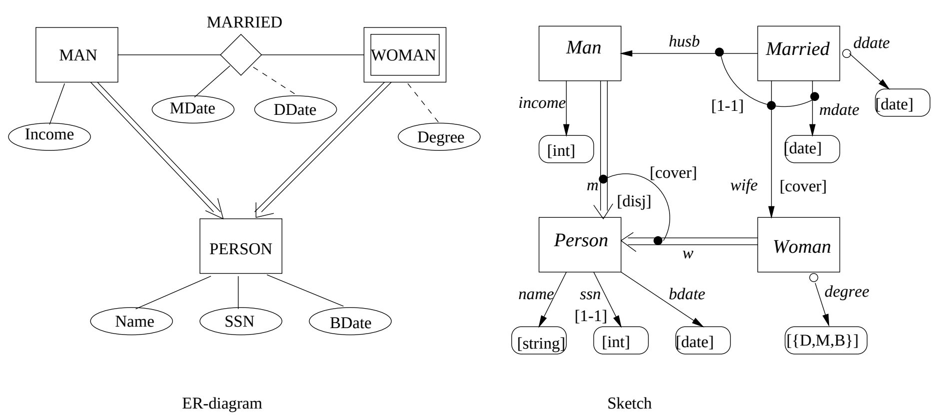


Fig. 1. Sketches vs ER diagrams

A rough view on the universe is described by the ER-diagram on the left side of Fig. 1 in a conventional ER-notation. The semantics of nodes and attributes is hopefully clear from their names, *MDate,DDate* are dates of marriage and divorce (the latter is optional). The domain of the optional attribute *Degree* is a set consisting of three values *D, M, B*. The class *Woman* is of weak entity type since users are assumed to be interested only in women who are or have been married.

Note, the ER-diagram does not reflect reality exactly: the diamond means that the pair (*wife,husb*) is an identifier (key, in database terms) whereas, in general, this 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*). The diagram is drawn as shown since in the notational ER-standard there is 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 right side of the figure. Note the [1-1] marker on the key attribute *ssn* and the cover marker on the arrow *wife*.

Rectangle nodes are *abstract classes* whose extensions should be stored in the DB. Oval nodes are predefined *value domains* whose semantics are *a priori* known to the DBMS. For the sketch approach, [`int`] and [{`D,M,B`}] 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 [`int`] its intended semantics is the predefined set of integers. At the same time, '*Person*' is a *name* labeling nodes without imposing any constraints. Generally speaking, we should also give names to the nodes labeled by [`int`] and [{`D,M,B`}], say, '*Number*' and '*Label*'. However, since the intended semantics of so marked nodes is fixed and remains the same for all schemas (independently of names of these nodes : '*Label*', or '*Tag*', or '*Attribute*' etc.), we adopt the convention of using such markers instead of names (they will be printed in the roman font). So, abstract class nodes are labeled by names, and value domain (printable) nodes are labeled by markers expressing their intended semantics (while their names become redundant and can be omitted).

3.4 Sketches vs. UML: simple examples

Let us consider the UML-diagram D_1 in Fig. 2(i), which models an association class (see, e.g., [17]). It follows from the informal explanation in [17] that *Job*-objects are nothing but pairs (c, p) with c a *Company*-object and p a *Person*-object, which themselves can have properties, *eg*, *salary, dateHired* and the like.

The corresponding sketch specification is presented on the right. Tails and heads of the horizontal arrows declare the same constraints as the left and right superscripts over the association edge on the UML-diagram (see the previous section). Marker [`inv`] hung on the couple of horizontal arrows denotes the predicate of being mutually inverse functions. Marker [`graph`] means that $[\![\, Job \,]\!]$ is the graph of (each of) these functions, that is, if $(c, p) \in [\![\, Job \,]\!]$ then $c = [\![\, employer \,]\!](p)$ and $p \in [\![\, employee \,]\!](c)$. Here $[\![\ ]\!]$ denotes semantic mapping assigning sets to nodes and functions to arrows.

One more example is presented in Fig. 2(ii). The UML-diagram D_2 models the so called *qualification* construct when to one end of an association edge two coupled

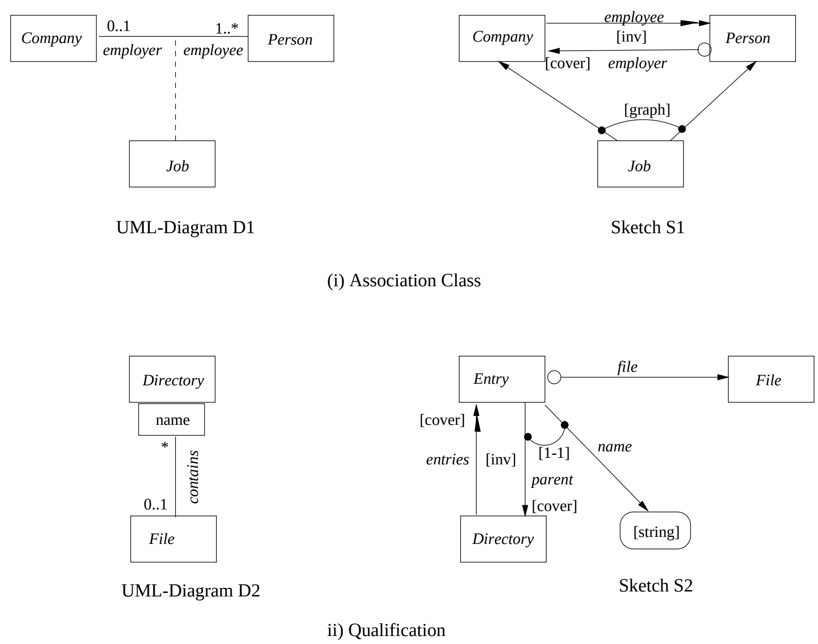


Fig. 2. Sketches vs. UML: two examples

nodes are attached: this means that in order to select an object on the other end of the association one should point out a value of the qualifier. In the example, *name* (smaller rectangle) is a qualifier and thus for a given *Directory*-object d, while $d.contains$ is the set of files in d, $d[\text{foo}].contains$ is the single *File*-object having name *foo* in the directory d.

A sketch specification of the same data semantics is presented on the right.

3.5 Sketching visual models: visualization aspects

In comparing given notational samples between themselves, the question of which one is "more right" is incorrect: any notation with unambiguously specified semantics can be used. However, the question of which notation is more clear and transparent w.r.t. its intended semantic meaning is quite reasonable.

Visualization aspects of sketching ER-diagrams. The sketch specification has a few noteworthy advantages over the ER-diagram.

In the first place, in the ER-diagram, labeling of a node sometimes indicates a set (e.g. MAN), and sometimes it indicates a mapping (e.g. Income is actually a mapping from men to integers). In the sketch specifications, nodes always specify sets, and arrows always specify mappings.

Secondly, the ER-diagram improperly visualizes diagram predicates. For example, a major part of the relationship semantics attributed to the node Married is nothing but the [1-1] property of the corresponding diagram (see sketch). In the ER-diagram this *diagram* property is attributed to the node. It's evidently a poor notation since normally it's assumed that a diamond node denotes relationship, that is, [1-1]-property, for entity nodes connected to it but actually it's not necessarily the case. In the given example, the triple-arrow span possesses [1-1] (as it's shown on the sketch) not the two-arrow span that is suggested by the ER-diagram.

The situation with declaring the node Woman a weak entity is even worse. Suppose, for example, that the same node occurs in another relationship (say, Employment between Woman and Company) where the corresponding role arrow is not a cover, that is, not all Woman-entities are employed. What should be the shape of the Woman node now: double-framed or ordinary? It is clear that the problem has occured since an arrow property is misleadingly attributed to the arrow source.

Finally, these improper visualizations give rise to artificial heterogeneity of nodes in ER-diagrams (relationships, entities, weak entities). As a consequence, since different users may view, and correspondingly specify, the same node in semantically different ways (e.g. one user may consider marriages to be a relationship, whereas another user sees it as an entity type in its own right), integration of different ER-diagrams with overlapping semantics becomes extremely difficult (see, e.g., [18]). In contrast, within the sketch framework the integration problem can be approached in a quite natural and effective way [7].

Visualization aspects of sketching UML-diagrams. Compare UML-diagram D_1 and sketch S_1 on Fig. 2(i), which express the same semantics. On the whole, the sketch presents a more detailed specification – all functions involved are shown explicitly – while the UML-diagram can be considered as a special abbreviation. In fact, in D_1 we have two different abbreviations. One is the presentation of two mutually inverse functions by one undirected edge each of whose ends carries the corresponding name and multiplicity constraint. It is a reasonable abbreviation, it makes graphical image more compact and multiplicity expressions are as good as special arrow heads and tails, or even better w.r.t. mnemonic efforts. Note, however, when one considers compositions of many references, undirected visualization can lead to mistakes. As for abbreviating the arrow span outgoing node *Job* in the sketch S_2 by an edge going into another edge in D_2, it is a purely syntactical trick hardly clarifying semantic meaning of the association class *Job*. In addition, such a way of presenting binary associations does not work for more than two-ary associations. In contrast, arbitrary n-ary association are presented by sketches by n-arrow spans in a uniform way.

Diagram D_2 in Fig. 2(ii) is an even more powerful abbreviation: four nodes and four arrows of the actual semantic picture (specified by sketch S_2 in detail) are compressed in a graphical image with only three nodes and one edge. However, the distance between visual schema D_2 and its semantic meaning ("congruently" specified by sketch S_2) is so large and meandering that diagram D_2 hardly can be considered as presenting a good visualization mechanism.

Summary. Of course, the issue of visualization is of complex cognitive nature and such culture-dependent points as notational habits and preferences, notions of convenience and elegance, and many others, can play significant role; analysis of such things goes far beyond the sketch formalism as such. Nevertheless, we believe that clear logical structure of sketch specifications and the presence of well-defined semantics for them make the sketch format a proper foundation for building a really good graphic notational system. In particular, it even allows the setting of a formal framework for the problem, for example, one may formulate the *commutativity principle*: visualization of conjunction of diagram predicates should be conjunction of visualizations. In general, we may think of organizing collections of specification and visualization constructs into similar mathematical structures and then visualization appears as a structure-preserving mapping between these collections, that is, their morphism. Actually, it gives rise to a consistent mathematical framework for building graphic notations. Close ideas were developed by Goguen and his group in their studies of reasonable graphical interfaces [13].

Finally, concerning the visualization of sketches, it must be emphasized that visualization on a computer (where one can use colour and dynamic highlighting of marked diagrams) is much clearer than visualization on paper.

4 Conclusion

The main question of the approach above is whether the specificational tools of a given visual model $\mathcal{M}$ can be converted into a sketch signature. In other words, whether the expressive power of the sketch language is sufficient to cover a wide class of visual models. The answer is positive and somewhat surprising: it follows from general results of category theory that any specification having a formal semantics can be replaced by an equivalent sketch in some signature. In fact, for one remarkable signature called the *topos signature* a universal modeling power was proven: any specification having a formal semantics can be replaced by a sketch in the topos signature (see,*eg*, [15] where the equivalence of toposes and higher-order logic is proven in a very transparent way). In other words, anything that can be specified formally can be specified by sketches as well. So, sketches do provide a real possibility to handle heterogeneity in a consistent way.

At the same time, by suggesting sketches we do not wish to force everyone to use the same universal graphical language – let one use that collection of graphical constructs one likes. What we actually suggest concerns *what should be specified* in graphical schemas (if, of course, one is concerned about their semantics, which we believe is almost always necessary). After that, the question is *how to visualize* the specification. An accurate distinction between logical specification and its syntactic-sugar visualization, and simultaneously their parallelism, are the two key advantages of the sketch approach from the practical view point.

A clear distinction between logical specification and its visualization is provided by the presence of formal semantics for sketches, and makes the sketch notation favorable in comparison with many notational systems used in software engineering. Of course, logic itself does not decide all of the problems: design of a good notational system is an art rather than technology. Nevertheless, the strict sketch pattern makes it possible to designate the formalizable part of the design task and state a precise basic framework for its consideration (section 3.5).In the paper we have occasionally touched on this issue of visualization but, of course, the problem needs a special elaboration; we leave it for future research.

Parallelism of specification and graphical visualization is provided by the graph-based nature of the sketch logic, and sharply distinguishes sketches from those visual models which are externally graphical, yet internally are based on predicate-calculus-oriented string logics. The repertoire of graphical constructs used in these models has to be bulky since all kinds of logical formulas require its special visualization. Configurations/shapes of these visualization constructs can be rather arbitrary because there are no evident natural correlations between graphics and logical string-based formulas. In particular, this problem is actual for the modern CASE-tools whose underlying logic is relational, that is, string-based.

Of course, there are situations when something can be easily described by a logical formula but hardly by a graphical image, and there are inverse situations as well. A good language has to combine graphical and string logics into a flexible notational mechanism.

On the whole, the sketch view we suggest gives rise to a whole program of refining the vocabulary of visual modeling, making it precise and consistent, and unified. In the sketch framework, *any particular diagrammatic notation appears as a particular visualization of the same common specificational format* – the format of sketches. Besides this unifying function, an essential advantage of the sketch format is the extreme brevity of its basic vocabulary exhausted by the three kinds of items: nodes, arrows and marked diagrams. Nevertheless, as it was discussed above, the sketch language is absolutely expressive and possesses a great flexibility.

Well, one might argue that given some particular discipline of VM, the virtual possibility of expressing its modeling constructs in the sketch language has nothing to do with effective usage of sketches in the field. This is very much true, and to build a discipline on the sketch ground one should thoroughly explain and specify its modeling constructs in the sketch framework. In effect, this is a research program to be fulfilled for each given VM-discipline but our current experience of applying sketches to real specificational problems in software engineering is promising: in cases where we applied sketches they appeared as a precise logical refinement of the existing notation rather than an external imposition upon it. In particular, in semantic data modeling sketches can be seen as a far reaching yet natural generalization of functional data schemas, ER-diagrams and OOA&D-schemas [14, 9], in meta-specification modeling they appear as essential generalization of schema grid developed by the Italian school (cf. [11] and [6]) and we expect that in process modeling sketches will be a natural development of interaction diagrams [1].

References

1. S. Abramsky. Interaction categories and communicating sequential processes. In A.W. Roscoe, editor, *A Classical Mind: Essays in honour of C.A.R.Hoare*, pages 1–15. Prentice Hall Int., 1994.

2. P. Atzeni and R. Torlone. Management of multiple models in an extensible database design tool. In *Advances in Database Technology – EDBT'96*, 5th Int.Conf. on Extending Database Technology, Springer LNCS'1057, pages 79–95, 1996.
3. A. Bagchi and C. Wells. Graph-based logic and sketches. In *10th Int.Congress of Logic,Methodology and Philosophy of Science, Florence, 1995*, Florence (Italy), 1997. Kluwer Acad.Publ.
4. M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall International Series in Computer Science, 1990.
5. J. Barwise and J. Etchemendy. A computational architecture for heterogeneous reasoning. In *Theoretical Aspects of Rationality and Knowledge*, pages 1–27. Morgan Kaufmann, 1998.
6. 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.
7. B. Cadish and Z. Diskin. Heterogenious view integration via sketches and equations. In *Foundations of Intelligent Systems*, Proc. 9th Int.Symposium, *ISMIS'96*, Springer LNAI'1079, pages 603–612, 1996.
8. P.P. Chen. The entity-relationship model – Towards a unified view of data. *ACM Trans.Database Syst.*, 1(1):9–36, 1976.
9. Z. Diskin. Variable set semantics for generalized sketches: Why ER is more object-oriented than OO. Technical Report 9603, Frame Inform Systems, Riga, Latvia, 1996. ftp://ftp.fis.lv/pub/diskin/ERvsOO.ps.
10. Z. Diskin. Formalization of graphical schemas: General sketch-based logic vs. heuristic pictures. In *10th Int.Congress of Logic,Methodology and Philosophy of Science, Florence, 1995*. Kluwer Acad.Publ., 1997. (On ftp: .../ PAPERS-Math/lmpsci95.ps.gz).
11. Z. Diskin. The arrow logic of meta-specifications: a formalized graph-based framework for structuring schema repositories. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Seventh OOPSLA Workshop on Behavioral Semantics of OO Business and System Specifications, OOPSLA98*, TUM-I9820, Technische Universitaet Muenchen, 1998. (ftp://cs.chalmers.se/pub/users/diskin/PAPERS-DB/oopsla98.ps).
12. Z. Diskin, B. Kadish, and F. Piessens. What vs. how of visual modeling: The arrow logic of graphic notations. In *Behavioral specifications in businesses and systems*. Kluwer, 1999.
13. J. Goguen. Semiotic morphisms. Technical report, University of California at San Diego, 1997. TR-CS97-553.
14. 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.
15. J. Lambek and P. Scott. *Introduction to higher order categorical logic*. Cambridge University Press, 1986.
16. G.W. Mineau, B. Moulin, and J.F. Sowa, editors. *Conceptual graphs for knowledge representation*. Number 699 in LNAI. Springer, 1993.
17. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
18. S. Spaccapietra, C. Parent, and Y. Dupont. View integration: a step forward in solving structural conflicts. *IEEE Transactions on KDE*, 1992.