Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Data & Knowledge Engineering 47 (2003) 1–59

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Variable set semantics for keyed generalized sketches: formal semantics for object identity and abstract syntax for conceptual modeling ☆

Zinovy Diskin *, Boris Kadish [1]

F.I.S. Group, Inc., Riga, Latvia

Received 23 January 1997; received in revised form 8 July 2000; accepted 5 December 2002

Abstract

We introduce a mathematical framework where a formal semantics for object identity can be built irrespectively to computer related things like object identifiers, memory allocations etc. Then, on this base, we build formal semantics for a few major constructs of conceptual modeling (CM) such as association, aggregation, generalization, *isA*- and *isPartOf*-relationships. We also give a formal meaning to the two fundamental dichotomies of CM: *objects* vs. *values* and *entities* vs. *relationships*.

On the syntactical side, the language we use for specifying our formal semantic constructs is graph-based and brief: specifications are directed graphs consisting only of three kinds of items—nodes, arrows and marked diagrams. The latter are configurations of nodes and arrows closed in some technical sense and marked with predicate labels taken from a predefined signature. We show that this format does provide a universal abstract syntax for the entire CM-field. Then any particular CM-notation appears as a particular visualization superstructure (concrete syntax) over the same basic specification format as above.
© 2003 Elsevier B.V. All rights reserved.

Keywords: Conceptual modeling; Object-oriented visual modeling; ER; UML; Object identity; Entity; Relationship; Association; Aggregation; Composition; Generalization; Formal semantics; Category theory; Variable set; Keyed sketch

☆ An early version was published in Object-Oriented and Entity-Relationship Modeling, Proceedings of the 14th International Conference, Springer Lecture Notes in Computer Science, vol. 1021, 1995, pp. 226–237.

* Corresponding author. Address: 15075 Lincoln Street #620, Oak Park, MI 48237, USA. Tel.: +1-248-968-9511.
E-mail addresses: zdiskin@acm.org (Z. Diskin), kadish@fis.lv (B. Kadish).

[1] Address: Frame Inform Systems Ltd., 31 Elizabetes Street, Riga LV-1010, Latvia. Tel.: +371-702-5300; fax: +371-702-5310.

0169-023X/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S0169-023X(03)00047-8

1. Introduction—why, what and how of the paper

Normally, an information system serves a fragment of the real world. A general prerequisite of its proper design and functioning is thus some representation mapping (maybe, implicit)

$$(*) \qquad r : \mathscr{W} \to \mathscr{C},$$

where $\mathscr{W}$ and $\mathscr{C}$ are models of, respectively, what we have in the world and in the information system ("computer"). The goal of conceptual modeling (CM) as a software engineering discipline is just in building frameworks where such mappings could be precisely specified.

A classical sample is the famous ER where $\mathscr{W}$-models are (extended) ER-diagrams and r is their transformation into relational schemas ($\mathscr{C}$-models). However, despite ER's great success in theory and practice, the fundamental difference between the languages used for specifying its $\mathscr{W}$- and $\mathscr{C}$-sides restricts its effectiveness. Briefly, because of that conceptual difference, much of valid information contained in $\mathscr{W}$-models is essentially transformed (if not lost at all) when passed to the $\mathscr{C}$-side and correlations between $\mathscr{C}$ and $\mathscr{W}$ become too complicated and obscured.

Object orientation (OO) has essentially contributed to resolving the problem. Its essential conceptual plus is that the world can be seen naturally as built from objects and then, within the OO-framework for the $\mathscr{C}$-side, the representation (*) becomes a natural mapping of real world objects into software objects in the computer. A typical sample here is transformation of a UML diagram ($\mathscr{W}$-side) into a collection of classes specified in some OO programming language ($\mathscr{C}$-side).

However, though in the OO framework the languages for the both sides are based on the same concept, some essential information contained in $\mathscr{W}$ is still lost or obscured in $\mathscr{C}$. For example, such $\mathscr{W}$-constructs as aggregation or composition or, say, the subtle differences between aggregations and associations, usually have incomplete (if any) implementations on the $\mathscr{C}$-side. Anyway, these implementations are necessary ad hoc because the $\mathscr{W}$-constructs above are not supported by the corresponding constructs of the programming language. An important exception is implementation of *isA*-relationships via inheritance but it seems to be the only exception.

An evident cause of this gap is the absence of precise semantics for $\mathscr{W}$-constructs just mentioned. Moreover, to date OO lacks even a framework where such semantics could be built. [2] So, the research program underlying the present paper is aimed, first of all, at building formal semantics for the $\mathscr{W}$-constructs mentioned above. More generally, we have in mind building some mathematical framework where formalized semantic meanings of EER-diagrams, [3] or UML diagrams, or one's favorite XXX-diagrams would become particular instances of the same abstract mathematical structure (like addition of integers or multiplication of non-zero reals are instances of the same abstract structure of group). A natural next step is to arrange the $\mathscr{C}$-side in a similar way so that representation (*) would be a structure-preserving mapping of similar mathematical structures (like group homomorphisms).

[2] To avoid misunderstanding, we emphasize once again that we talk about the $\mathscr{W}$-side of OO. As for the $\mathscr{C}$-side, the basic properties of OO (object identity, encapsulation, inheritance and others) are well elaborated and provided with semantics (sometimes even formal, see e.g. [19,28] for references to the two prominent formal semantic frameworks: *coalgebra* and *hidden algebra* respectively).

[3] Here and below EER refers to some extension of the classical ER-vocabulary with additional modeling constructs; a few such extensions were proposed in the literature.

Note also the shift from syntax to semantics in understanding (*). In the conventional view, r works on the level of syntax, transforming specifications (EER-, UML-, XXX-diagrams) into specifications (pieces of code). For us, (*) is first of all a morphism of mathematical structures modeling semantic meaning of specifications in question; the corresponding transformation of the very specifications is then a derived issue.

On a whole, it is a large program with a large agenda; we hope that upon its completion the discipline of CM will have a level of precision and mathematical baseness comparable with that we find in mechanical and electrical engineering. Some details on the paper's content are shortly described below.

1.1. What is done in the paper

1.1.1. Semantics

We introduce a mathematical framework where a formal semantics for object identity (as opposed to object state) can be built irrespectively to $\mathscr{C}$-related things like object identifiers, memory allocations etc. On this base we then build a formal semantics for major constructs of CM such as association, aggregation, generalization, *isA*- and *isPartOf*-relationships.

We also give a formal meaning to the two fundamental dichotomies of CM: *objects* vs. *values* and *entities* vs. *relationships*, and also to somewhat subtle differencies between aggregation and association and between specialization and generalization. In this way our framework provides formal semantics for main CM-languages such as ER-diagrams (with their enhancements of various types) or UML class diagrams (their structural part) as well. [4]

1.1.2. Syntax

The language we use for specifying our formal semantic constructs is graph-based and extremely brief—specifications are directed graphs consisting of only three kinds of items:

- *nodes* (denoting sets or set-like objects),
- *arrows* (mappings between them),
- *marked diagrams* (predicate declarations), that is, configurations of nodes and arrows closed in some technical sense and marked with predicate labels taken from a predefined signature.

Following mathematical category theory (CT), we will call such specifications *sketches*. [5] Our sketches are an essential generalization of original categorical sketches towards more flexibility

[4] Declarations of having built formal semantics for EER-diagrams of some kind are not seldom in the literature. However, we are not aware of formal semantics for EER capturing the dynamic dimension of object identity and advanced CM-constructs like aggregation or composition. At the same time, we show in the paper that dynamic aspects are principle for semantics of these constructs and thus any formalization disregarding them is essentially incomplete.

[5] CT is an algebraic discipline that has greatly influenced, and continues to influence, the language and conceptual framework of modern mathematics and theoretical computer science. There are areas where the high (and even unique) suitability of CT language and machinery is commonly recognized, in particular, this is the case for meta-mathematical studies, especially for the part that might be called *mathematical structure engineering* (see [14] for a detailed discussion). We will often use the adjective *categorical* in the meaning of being based on, or developed with, CT-methods and ideas.

Sketches were invented in CT for specifying mathematical structures in a graphical yet precise formal way (see [2] for an exposition and references to the literature).

and convenience for CM-applications, their formal presentation and comparison with original categorical sketches can be found in [10].

1.1.3. Unifying specification framework for CM

A surprising result about sketches proven in CT is that *any* formal construction, X, can be specified by a sketch $\mathsf{S}(X)$. [6] We will refer to this procedure as *sketching* X and write $\mathsf{S}(X) = \mathbf{ske}X$. For an informal construction $\mathscr{X}$, we will also say "sketching $\mathscr{X}$" and write $\mathsf{S}(\mathscr{X}) = \mathbf{ske}\mathscr{X}$, which actually means sketching a certain formal refinement X of $\mathscr{X}$. Of course, in the latter case, discovering and building X is the most non-trivial part of the entire sketching procedure.

So, our sketches combine graphical and compact syntax, on one hand, and extreme semantic expressiveness on the other. They themselves can be used as a convenient CM-language, and even in an industrial environment too. However, we would like to stress from the very beginning that what we propose with sketches is not one more CM-language (with these and those advantages) but a framework that may well serve as a *universal abstract syntax* for the entire CM-field. Such an ambition is based on the following two considerations.

The first one is that because of the absolute expressiveness of the sketch language, semantic meaning $\mathscr{M}(\mathsf{D})$ of any diagram D in any reasonable CM-language (EER, ORM, UML, whatever) can be expressed by a sketch $\mathsf{S}(\mathsf{D}) = \mathbf{ske}\mathscr{M}(\mathsf{D})$, provided $\mathscr{M}(\mathsf{D})$ can be described in formalizable terms.

The second consideration is an empirical fact, and the paper provides some evidence for it. The fact is that for a consistent diagrammatic notation practically used in CM like various versions of ER-diagrams or the UML, sketch $\mathsf{S}(\mathsf{D})$ specifying semantics of diagram D is normally more or less *graphically similar* to D. Actually, it is not accidental and there are deep reasons for this similarity. Briefly, it is because the sketch framework assumes *set-based* (we will also say *setwise*) semantics and *graph-based* syntax, which are the two major general peculiarities of current CM-languages; we will discuss the issue in more detail in Section 5.1.

Owing to the graphical similarity between a diagram D and its semantics' sketch S, the former can be considered as a special visual abbreviation of the latter. In other words, a reasonable CM-notation appears as a particular visualization superstructure (*concrete syntax*) over the same basic sketch format (*abstract syntax*). In this way the sketch language provides a unique unifying potential reducing the diversity of CM-languages and notations to the much more manageable diversity of just (syntactic-sugar) visualizations over the same specification format (see also [17] for this issue).

So, the sketch language can well play the role of specificational esperanto for CM and hence be aimed at the same goal as the UML, the approach is however entirely different and semantically well-based. If one considers the UML as a project in progress rather than a final result (a shift that seems to be happening right now), then sketches do not compete with the UML, just the opposite: they appear as a solid and mathematically justified foundation for the entire UML building and

[6] What does it mean to be a *formal construction* is a special question. Answering it in details involves some issues in foundations of mathematics and needs a certain philosophy. Briefly, in the view accepted by modern mathematics, to be *formal* means, after all, to be expressible in the language of formal set theory. The CT result referred above just states that models of formal set theory can be specified by sketches as well.

provide conditions for its future proper development (see [13,15] for a sketch account of the UML class diagrams).

1.2. How it is done—conceptual modeling via sketches

Sketches are just a language for specifying constructions or universes built from set-like objects and mappings between them. The procedure of sketching a construct χ amounts to presenting χ in terms of sets and mappings. Actually, a certain experience of building such presentations is already accumulated in CM/OOAD because these disciplines are also based on general premises of setwise semantics and graph-based syntax. Still, in the "rarified atmosphere" of the sketch specifications—only nodes, arrows and diagram predicates—a CM-specialist would feel himself not very comfortably. Indeed, a typical universe of discourse considered in CM normally includes many constructs and concepts like, say, aggregations, specialization, composition or the object-value dichotomy which seem to be far beyond the pure sets-and-mappings pattern. As was mentioned above, formal semantics for these constructs can be specified by sketches in principle but it does not give any clue for

(i) how to build this formal semantics, and after that,
(ii) how natural and convenient are such sketch specifications from the software engineering viewpoint. [7]

The main result of the present paper is in exhibiting set-based formal semantics for major CM-constructs thus answering to (i), and demonstrating (we believe) suitability of sketch conceptual models w.r.t. the (ii) criterium. We achieve this goal in three main steps.

Step A: object classes are sets. In Section 2 we apply the sets-and-mappings pattern to specifying CM-constructs and achieve some partial success in modeling associations between object classes, specialization/generalization of classes and of associations, and in explicating distinction between value domains and object classes. We also show that internal type structure T (tuple, variant, grouping) of a set X can be well treated externally in the following way. Set X is considered plain and homogeneous but a certain predicate $P(T)$ is declared for an arrow diagram $D(T)$ adjoint to X, and it *entails* T-structure for X's elements. Thus, type structure of X's elements is implicit and derived rather than declared explicitly.

However, more complex examples and analysis reveal essential inherited limitations of the static set framework for building adequate semantic foundations for CM.

Step B: object classes are variable sets. In Section 3 we elaborate the framework of *variable* sets-and-mappings semantics for sketches. Variable set is a set whose extension changes in time and any two (logically related) extensions are connected by binary *same*-relations formally explicating identity of objects of this set. So, a state of variable set is a set of *object states* rather than of *objects* themselves. The latter are trajectories determined by the *same*-relations in the state space.

[7] A major problem with building formal mathematical models for engineering constructs is that often they either model just a very limited reduct of the construct in question or/and are so complicated that engineers totally lose interest in them. Another alternative also not rare in the literature is a kind of quasi-formal modeling that is evidently even worse.

Various types of *same*-relations correspond to various object ontologies and in this way we formally explicate dichotomies of *objects* vs. *values* and *entities* vs. *relationships*.

Step C: object classes are keyed variable sets. Finally, in Section 4, semantic relationships between object classes (like aggregation, specialization, composition) are formally explicated as derivability of one class' *same*-relations from *same*-relations of other classes. In more detail, an ontologically dependant class C has to have a set of arrows $f_i : C \to D_i$ $(i = 1, \ldots, n)$, going from C to classes on which identity of C-objects depends and C's same-relations are derived from D_i's *same*-relations with mappings f_i. It is natural to call the set $\{f_i | i = 1, \ldots, n\}$ the key to the class, and it is indeed a sort of generalization of the well-known in the relational model notion of primary key. So, sketches for conceptual modeling are keyed sketches: each node denoting an object class is endowed with a key and interpreted by a variable set with same-relations either set independently or derived as specified by the key.

To summarize, variable set semantics for keyed sketches is a precise mathematical framework where the "world" side of CM-mapping $r : \mathscr{W} \to \mathscr{C}$ can be formally explicated (in a OO-way). The essence of the framework could be briefly summarized by the thesis: The *world consists of sets and mappings which are essentially changeable and ontologically connected*. Actually, it is a categorical thesis and the machinery we introduce and use in the paper is categorical in its nature. However, we do not use any technical CT-results and all our references to CT are references to the conceptual framework and general inspiration only (a few more details can be found in Section 5). So, technically the paper is selfcontained and actually uses only elementary mathematical notions like those of set and mapping between sets.

In addition to the three main sections-steps just described, in Section 5 we place the approach in a more general mathematical context and, on the other hand, relate it to well-known frameworks developed in CM.

Table of contents below gives a more detailed presentation of what can be found in the paper. In addition to the section–subsection structure shown in Table, subsections are divided into small enumerated units (many of which but not all are entitled) to ease references and guidance through the text.

1.3. What is outside of the paper

1.3.1. Query languages in the sketch framework

The both sides of the mapping (*) are universes carrying algebraic operations (queries, in the database context). In building a particular conceptual model $\mathscr{W}$, an important choice one has to make is to decide which items in the model are to be considered basic and which are derived. A reasonable model of practically any universe of real interest involves derived items, correspondingly, on the level of abstract sketch syntax, operations over sketches. The latter are *diagram operations* whose input and output are specified by arrow diagrams.

Diagram algebra, with formal definitions of syntax and semantics for diagram operations, is a technically somewhat bulky discipline presented in [10]. In the paper we will describe it very briefly and informally, and show its usage in CM in a number of examples. As for the issue of query languages over sketches, that is, setting a collection of diagram operations and considering its expressiveness (completeness w.r.t. one or another criteria), for the purposes of the present paper it is quite another story and we will not touch upon it.

1.3.2. Behavior aspects of semantic relationships

The *isA*-relationship between object classes has the two sides: structural, or data inheritance, and behavioral, or procedure (method) inheritance, which are well-known on the $\mathscr{C}$-end of OOAD. Similarly, there are the two sides of other CM-constructs like aggregation, composition etc. In the paper we will deal with only the structural side of the constructs leaving the behavioral side for a companion paper (to appear elsewhere, see Section 6 for a bit more of details). Note however that in our framework structural properties are not static: our objects are changeable and, say, update semantics can be well captured in our framework.

2. Categorical sketches — The world consists of sets and mappings...

In this section we investigate the possibilities and limitations of the specification paradigm italicized in the title. Though it may seem too restrictive for CM, we will show that with a proper treatment it does provide a rich semantic framework where many important constructs of CM can be given an adequate formal semantic meaning, including complex types, associations between object classes (and generalization of associations) and some aspects of the fundamental distinction between object classes and value domains. However, we will also see in the last subsection that still there are a few important concepts and constructs in CM, semiformally formulated in, say, ER and UML, which cannot be captured in the static set framework.

2.1. Conceptual modeling via sketches—getting started

We begin with applying sketches to specifying a very simple and easy for set-theoretical formulation universe. Its semiformal description is given by the UML class diagram D in Fig. 2(a). Nodes denote object classes and vertical arrows between them denote generalization/specialization relationships. Horizontal lines between nodes denote associations between classes, note also multiplicity constraints on the association ends (we use [37] as a standard reference to the UML: notation, terminology and explanations). Names on nodes and association ends make semantics of the associations clear; as for the middle arrow between the association lines—the so called association generalization [37, p. 163]—it simply says that a man can drive a car only if he owns it.

Our goal is to specify the universe in entirely formal terms of abstract sets and mappings. For a while we will restrict ourselves with the static set setting just to introduce the sketch language and show how it can work. We begin with entering some notation and terminology on mappings between sets—the major ingredient of the sketch language.

2.1.1. Mappings between sets—notation and terminology

Given sets X, Y, we assume a general mapping between them $f : X \to Y$ to be partially defined and multi-valued. The *domain* of f is the set $\mathrm{Dom}(f) \subseteq X$ of all $x \in X$ for which $x.f$ is defined, and if $x \in \mathrm{Dom}(f)$ then $x.f$ is a non-empty subset of Y. Here we write $x.f$ for the value of f at the argument x, we will also write $f(x)$ and sometimes $f.x$ for that (it will be always clear from the context what is the mapping and what is the argument). If f is a priori known to be single-valued then we consider $x.f$ an element of Y rather than a singleton subset of Y. The sets X and Y will be called the *source* and the *target* of f and denoted by $\Box f$ and $f\Box$ respectively. We call an arrow f

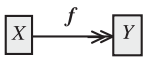
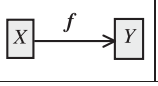
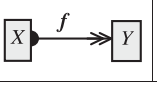
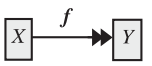
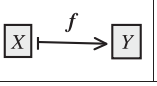
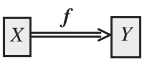
Semantics	f is partial and multi-valued	f is single-valued	f is total	f is covering	f is injective (or 1-1, monic)	f is inclusion
Notation						

Fig. 1. Properties of a general mapping $f : X \to Y$.

totally-defined or just *total* if $\mathrm{Dom}(f) = \Box f$. Dually to the notion of domain, the *range* of f, $\mathrm{Rg}(f)$, is defined to be the set of all $y \in Y$ s.t. $y \in x.f$ for some $x \in \mathrm{Dom}(f)$. Sometimes we will also write $f(X)$ for $\mathrm{Rg}(f)$ and call it the *image* of X under f. We will call f a *cover* if $\mathrm{Rg}(f) = f\Box$, this property is dual to being totally defined.

A single-valued mapping is called *injective* or *1–1*, or else *monic* [8] if for any two different elements $x, x' \in \mathrm{Dom}\ f$, $x.f$ and $x'.f$ are also different.

Finally, if we have two mappings $f : X \to Y$ and $g : Y \to Z$ such that $f\Box = \Box g$, then f and g can be composed into a mapping $(f \fatsemi g) : X \to Z$ defined in an evident way: for any $x \in \mathrm{Dom}(f)$, $x.(f \fatsemi g) \subset Z$ is defined to be the union of all sets $y.g$ for y running over the set $x.f \cap \mathrm{Dom}(g)$. Sometimes we will denote composition $f \fatsemi g$ by $f; g$.

A special kind of mapping is *inclusion*: if a set A is a subset of set X then there is an inclusion mapping $i : A \to X$ which does nothing: $x.i = x$ for any $x \in A$. Note, while $\mathrm{Dom}(i) = \mathrm{Rg}(i)$ for any inclusion but $\Box i \neq i\Box$ (unless A is X). That is why, though procedural "bodies" of inclusion mappings are trivial, they play an important role in functional diagrams by matching the sources and targets of the arrows to be composed and thus their explicit presence in functional diagrams is a must. We observed a few unnecessary complicated constructions in CM caused by disregarding of inclusion arrows. [9] Note also that in general we do not assume inclusions to be totally defined.

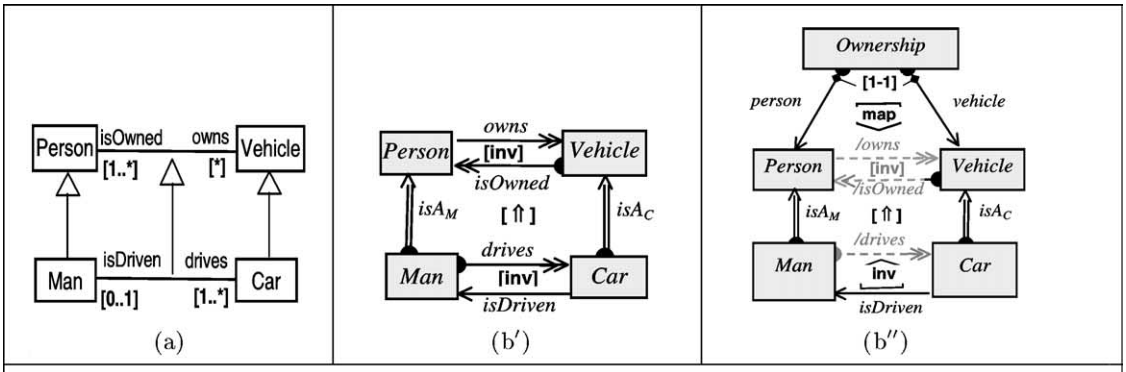
To denote these properties of mappings in sketches, we need some consistent graphic notation, a suitable variant is presented in Fig. 1. Its advantage is that a combination of arrow properties is visualized by the corresponding combination of visualizations. For example, a total single-valued cover is visualized by an arrow ◗⟶◗, and total monic is visualized by ◗⟼>. So, a useful principle of *visual superposition* holds: visualization of predicate superposition $P_1 + \cdots + P_n$ is superposition of the corresponding visualizations:

$$\mathbf{vis}(P_1 + \cdots + P_n) = \mathbf{vis}P_1 + \cdots + \mathbf{vis}P_n.$$

Now we return to our universe in Fig. 2.

[8] We borrow this term from CT.

[9] A characteristic example can be found in [5] where the authors consider conceptual schemas consisting of sets and partial functions between them. However, they do not distinguish between, for example, mappings $f : X \to Y$ and $f' : A \to Y$ with $A \subset X$, and even denote them by the same letter. (This improper equalizing different arrows like above is unfortunately somewhat traditional for CM, references to earlier papers with similar drawbacks can be also found in [5]). The authors then explore the operation of schema integration/merging but come to a surprising result that it is non-associative: $(S_1 + S_2) + S_3 \neq S_1 + (S_2 + S_3)$. To correct this evidently strange behavior, they need to introduce artificial concepts and means essentially complicating the algebra of schema merging. However, in a proper arrow framework, schema merging is at once seen to be an associative diagram operation (as it should be, of course, a bit more details are presented in Appendix B.4).



Predicates			
Name	Arity shape	Visualization (marker)	Semantics in **Sets**
inverse arrows	X, Y, f, g	[inv]	for any $x\in X, y\in Y$ $y\in x.f \Leftrightarrow x\in y.g$
semi-commutativity	A, Y, X, P, g, f, u, v	[⇑]	$u;v \le f;g$, that is, $Dom(u) \subseteq Dom(f)$ and for any $x\in Dom(u)$, $\{p.v \mid p\in x.u\} \subseteq \{a.g \mid a\in x.f\}$
jointly monic (separating) pair of mappings	X, f_1, f_2, Y_1, Y_2	[1-1]	for any $x,x'\in X$, if $x.f_1=x'.f_1$ and $x.f_2=x'.f_2$, then $x=x'$ (see sect. 2.3 for more general presentation)
Operations			
inversion of mapping	X, Y, f, /g	inv	for any $y\in Y$, $y./g := \{x\in X \mid x.f \ni y\}$, in other words, $/g = f^{-1}$
functional presentation of relation	R, p, [1-1], q, /f, X, [inv], Y, /g	map	for any $x\in X$, $x./f := \{r.q \mid r.p=x\}$ and for any $y\in Y$, $y./g := \{r.p \mid r.q=y\}$, in other words, $/f = p^{-1}; q$ and $/g = q^{-1}; p$

(c)

Fig. 2. Specifying via sketches: an example. (a) UML class diagram, D. (b′) Sketch $\mathsf{S}'_{\mathrm{sta}} = \mathbf{ske}M^{PL}_{\mathrm{sta}}(\mathsf{D})$ and (b″) Sketch $\mathsf{S}''_{\mathrm{sta}} = \mathbf{ske}M^{DB}_{\mathrm{sta}}(\mathsf{D})$, where $M^{PL}_{\mathrm{sta}}[\mathsf{D}]$ and $M^{DB}_{\mathrm{sta}}[\mathsf{D}]$ are formal refinements of the intended meanings of diagram D in the static set framework in the context of programming languages and databases respectively. (c) Signature of diagram predicates (besides arrow predicates specified in Fig. 1) and diagram operations used in $\mathsf{S}'_{\mathrm{sta}}$ and $\mathsf{S}''_{\mathrm{sta}}$.

2.1.2. Sketching the universe

We have four abstract sets *Person*, *Vehicle*, *Man* and *Car*, whose elements are called, respectively, *persons*, *vehicles*, *mans* and *cars*. The latter two sets are subsets of the former two and thus we have two inclusion mappings isA_M and isA_C as presented by sketch S'_{sta} in Fig. 2(b′). Index "sta" is added to remind that we search for formal meaning of the UML-diagram in the static set framework. Formal meaning of the association links in our example is clear: we have two pairs of mutually inverse mappings. Since some multiplicity constraints are supposed to hold for the associations as specified on the UML diagram, the mappings have special properties denoted by the corresponding special tails and heads (explained in Fig. 1).

The fact that the two mappings constituting an association are mutually inverse is expressed by declaring the corresponding predicate [inv] for them, that is, for the diagram consisting of the corresponding pair of arrows (top row in Fig. 2(c)). Such a declaration is visualized by "hanging" the predicate label [inv] on the diagram. Two such labels in the sketch S'_{sta} mean that it contains two such declarations:

$$[\texttt{inv}](\mathit{owns}, \mathit{isOwned}) \text{ and } [\texttt{inv}](\mathit{drives}, \mathit{isDriven}). \tag{1}$$

Note also that special arrow heads, tails and bodies used in the sketch are nothing but predicate markers/labels hung on the arrows.

Finally, the last component of the sketch S'_{sta} is the predicate label [⇑] hung on the outer arrow rectangle diagram:

$$[\Uparrow](\mathit{isA}_M; \mathit{owns}; \mathit{drives}; \mathit{isA}_C), \tag{2}$$

which declares it to be *semi-commutative*, that is, satisfying the condition

$$m.\mathit{drives}.\mathit{isA}_C \subseteq m.\mathit{isA}_M.\mathit{owns}$$

for each $m \in \mathit{Man}$. (Here expression $m.\mathit{drives}.\mathit{isA}_C$ denotes the set $\{c.\mathit{isA}_C | c \in m.\mathit{drives}\}$ and similarly for $m.\mathit{isA}_M.\mathit{owns}$). Using the operation $_\!\!\rightarrowtriangle\!\!_$ of mapping composition, the condition can be written in the form

$$\mathit{drives} \rightarrowtriangle \mathit{isA}_C \leqslant \mathit{isA}_M \rightarrowtriangle \mathit{owns}$$

which explains the term semi-commutativity. [10]

2.1.3. Some details

In the description above we have used the same names for both a syntactical item and its semantic meaning. It is a common, and reasonable, practice but at least for the first time of building sketch specification it would be useful to phrase semantics a bit more accurately.

Names in the sketch S'_{sta} in Fig. 2(b′) denote syntactical items: nodes, arrows and predicate labels hung on certain diagrams. Semantics of the sketch is a mapping, say, $[\![_]\!]$, which sends nodes to finite sets and arrows to partial multi-valued mappings between them so that incidence between node and arrows is preserved. For example, the arrow *owns*: *Person* → *Vehicle* is sent to a mapping $[\![\mathit{owns}]\!]$ of set $[\![\mathit{Person}]\!]$ to set $[\![\mathit{Vehicle}]\!]$. Of course, markers hung on the arrows must be respected,

[10] Commutativity predicate assumes equality instead of inequality (inclusion) and in our example would mean that a man drives a car only if he owns it and drives each car he owns.

for example, the mapping $[\![drives]\!]$ must be total and the mapping $[\![isA_C]\!]$ must be an inclusion. This latter condition means, in particular, that the set $[\![Car]\!]$ is a subset of $[\![Vehicle]\!]$.

In addition, other predicate declarations (marked diagrams) occurring in the sketch must be also satisfied by $[\![_]\!]$, that is, mappings $[\![owns]\!]$ and $[\![isOwned]\!]$ as well as $[\![drives]\!]$ and $[\![isDriven]\!]$ must be mutually inverse, and the diagram $([\![isA_M]\!], [\![owns]\!], [\![drives]\!], [\![isA_C]\!])$ must be semi-commutative. If all these conditions embodied into the sketch $\mathsf{S}'_{\mathrm{sta}}$ are satisfied by the mapping $[\![_]\!]$, it provides a *valid instance* of $\mathsf{S}'_{\mathrm{sta}}$. In fact, a valid instance *is* a mapping $[\![_]\!]$ like above.

Our description has become more accurate but still informal since we have note specified precisely what does it mean to "hang a predicate label on a diagram in a sketch". A formal presentation can be found in [10] but it seems to be not necessary for the present paper; some essential details are nevertheless presented in Appendices.

2.1.4. Remark—visualization vs. specification

It is instructive to compare the UML diagram D and the sketch $\mathsf{S}'_{\mathrm{sta}}$ in Fig. 2: the former appears as a special visualization of the latter. In particular, two special visualization constructs are used in D: (i) a line segment between object classes abbreviates a pair of mutually inverse arrows, and (ii) a special triangle-head arrow (between associations) actually denotes the semi-commutativity predicate.

The first visualization construct is apt: it replaces three items (two arrows and the marker) by one item in a natural and clear-how-to-deabbreviate way. Indeed, the UML encourages naming association ends, and these names are nothing but names of the arrows hidden in the association. As for the visualization construct (ii), its disadvantage is that while left and right triangle-head arrows denote actual mappings (inclusions), a similar middle arrow denotes a predicate label rather than a mapping. Such a notation can be misleading.

2.2. Arrow diagram operations and conceptual modeling

A diagram operation is an operation whose input and output are specified by arrow diagrams. Such operations are important and actually widespread (although often implicit) in CM. For example, let us reconsider our *Person–Vehicle* universe in another context.

2.2.1. Inverse arrows—predicate and operation

The situation when binary associations can be traversed in both directions as specified on the UML diagram D and sketch $\mathsf{S}'_{\mathrm{sta}}$ in Fig. 2(a) and (b′), is quite natural for programming but a very special one in the database context. In the latter, nodes represent relational tables and if we include the reference (a foreign key column) *isDriven* from the table *Car* to the table *Man* into the database schema, then the inverse reference should *not* be, in general, included into the schema. Otherwise we would include derived information (a query result) into the schema, which complicates database update procedures. However, it may happen that this derived arrow drives occurs into some view to the schema useful in applications. [11] In such a case, the user should have an augmentation of the schema with derived items including the arrow *drives*, where for each

[11] See [6,16] for specifying views in the sketch framework.

derived item an operation (query) producing it is specified. For the arrow *drives*, it is the operation of mapping inversion (see Fig. 2(c)) applied to the arrow *isDriven* as specified in the sketch $\mathsf{S}''_{\text{sta}}$.

Syntactically, a diagram operation looks like a special diagram predicate whose arity shape has a designated subshape called the *input arity*; other items in the arity shape, that is, outside the input shape, are called *output* or *derived*. For example, the input shape for the operation [inv⟩ is just an arrow together with its source and target. The entire arity shape is the augmentation of the input with an arrow going into the opposite direction. To distinguish derived items from basic, the former are shown with dashed lines and grey color, and their names are prefixed with slash / (we borrow the latter convention from the UML). The very label of diagram operation is bracketed asymmetrically by [inv⟩ in contrast to the predicate [inv]. As for the operation as such, its body is a procedure of computing the inverse to a given mapping, it can be implemented in any way matching the semantic specification in the last column of Fig. 2(c).

2.2.2. Remark

Of course, the inverse arrow operation is graphically very simple and a linear notation for it, say, $g = f^{-1}$, is practically unambiguous because of the well-known context: f^{-1} goes between the same nodes in the direction opposite to f. In general, however, diagram operations may have more complex input configurations consisting of a few nodes and a few arrows, and may produce a few new nodes and arrows. For such operations their diagrammatic nature is much more essential and linear notation for them may be ambiguous or bulky and awkward.

2.2.3. Diagram operations and constraints

Derived items, hence, operations producing them, often occur in conceptual models because of the necessity to specify constraints involving derived items. Consider, for example, our *Person-Vehicle* universe again in the database design context. Then it is natural to model the many-to-many association between *Person* and *Vehicle* by a relation rather than the pair of inverse mappings. So, we enter into the model a new set *Ownership* with two total single-valued projection mappings *person* and *vehicle* (see sketch $\mathsf{S}''_{\text{sta}}$ in Fig. 2(b″)) and hang on the span diagram the [1–1]-marker that makes *Ownership* a binary relation: below in Section 2.3.1 we will explain semantics of such a marker. Now the arrows *owns* and *isOwned* can be derived by applying to the span diagram the operation [map⟩ specified in the bottom row of Fig. 2(c). Note, the operation [map⟩ derives a pair of arrows together with the marker [inv] declaring them to be mutually inverse.

The presence of the derived pair (*owns*, *isOwned*) in the sketch $\mathsf{S}''_{\text{sta}}$ is useful for not only explicating the correspondence with our previous models (sketch S'_{sta} and the UML diagram D). In addition, it allows us to state two important constraints to the model: totality of the mapping/*isOwned* and semi-commutativity of the diagram $(isA_M, /owns, /drives, isA_C)$. Of course, these constraints could be reformulated without involving derived items by considering them as properties of just the relation (*Ownership*, *person*, *vehicle*) and the mapping *isDriven*. However, their formulation using derived items as presented on sketch $\mathsf{S}''_{\text{sta}}$ is much easier to comprehend and hence is much more suitable for a high-level conceptual model.

Situations we considered above are typical for CM. In addition, one often deals with diagram operations whose diagrammatic nature is hidden under an interface based on linear formulas written in strings; we will call such notations and interfaces *string-based*. For example, well-known operations of relational algebra are, in fact, diagram operations: it becomes clear if relations are

specified as sets with attached (1–1)-families of mappings as mentioned above and described below in Section 2.3.1. So, SQL appears as a string-based interface to an essentially arrow-diagram-based query language.

Thus, diagram algebra should be an essential component of CM/OOAD. However, foundations of diagram algebra, particularly accurate syntax for diagram operations, still have not been presented in the literature. A system of main mathematical definitions is developed in [10] and its practically oriented presentation will appear elsewhere. As for the present paper, we will use diagram operations in a semi-formal way.

2.3. Complex types by arrow diagram predicates

The example we have considered above demonstrates how sketches work for specifying plain sets and constructs over them but the reader may have doubts of whether sketches are equally good for managing complex types like tuple or grouping. Indeed, a basic prerequisite of the sketch approach is to interpret nodes as plain sets of unstructured elements and it may seem that complex types are foreign for, or at all beyond, the sketch language. Nevertheless, complex types can be quite naturally managed in the sketch framework.

2.3.1. Tuple type

Given a family of sets $Y_1, \ldots, Y_n$, instead of saying that a set X consists of n-tuples built from elements of the family, one can equivalently say that there is a *separating* family of total single-valued mappings, $f_i : X \to Y_i$, $i = 1, \ldots, n$, that is, a family satisfying the following condition:

[1–1] for any $x, x' \in X$, $x \neq x'$ implies $x.f_i \neq x'.f_i$ for some $i = 1, \ldots, n$

Indeed, in such a case the tuple-mapping $f = \langle f_1, \ldots, f_n \rangle$ into the Cartesian product of domain sets,

$$f = \langle f_1, \ldots, f_n \rangle : X \to Y_1 \times \cdots \times Y_n, \quad x.f \stackrel{\text{def}}{=} \langle x.f_1, \ldots, x.f_n \rangle$$

is also total, single-valued and one–one so that elements of X can be considered as unique names for tuples from a certain subset of $Y_1 \times \cdots \times Y_n$, namely, the image of f. In a certain context, elements of X can be identified with these tuples so that X is a relation up to isomorphism. In the classical ER-terminology [7], if the domains Y_i's are entity sets then f_i's are *roles* and any $x \in X$ is a relationship between entities $x.f_1, \ldots, x.f_n$.

The property of mapping f above to be one–one explains naming separating families *jointly monic* or *jointly* 1–1, and denoting them in sketches by the marker [1–1].

Since, conversely, for any relation the family of its projection mappings is separating, the very notions of a tuple set and a jointly monic family of mappings are equivalent. Correspondingly, on the syntax level, to specify a node as a relation one may leave the node without any marking but label instead the corresponding source of outgoing arrows by marker [1–1] (Fig. 3(a)). In fact, this is nothing but a well known idea of designating a key to relation and we will often call a separating family of arrows also a key.

So, the sketch in Fig. 3(a) specifies the node X as a ternary relation. Indeed, the intended semantics of this sketch is an extension mapping $[\![_]\!]$, which assigns sets $[\![X]\!]$, $[\![Y_j]\!]$ and mappings $[\![f_j]\!]$ to the corresponding nodes and arrows ($j = 1, 2, 3$) in such a way that the constraint expressed by

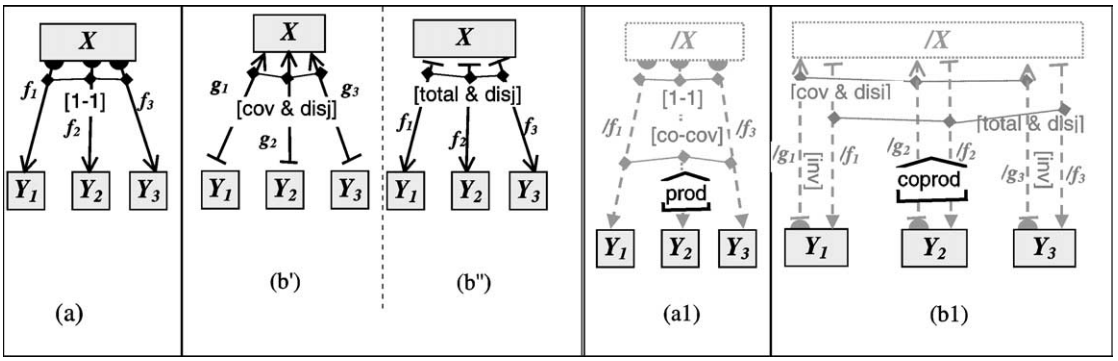


Fig. 3. Complex types via sketches. (a) Tuple type; (b′) and (b″) subvariant type in two equivalent presentations: $f_i = (g_i)^{-1}$; (a1) product operation; (b1) coproduct operation.

the predicate marker [1–1] is satisfied, that is, the family of mappings ($[\![f_j]\!]$, $j = 1, 2, 3$) is separating. Then, as it was shown above, $[\![X]\!]$ is a relation up to isomorphism.

2.3.2. Remark

Modelling N-ary relations by jointly monic families of arrows as above at once resolves the infamous problem of multiplicities for N-ary associations—details can be found in [15].

Our next goal is to sketch the familiar notion of *variant type* also called *disjoint union* or *coproduct*. Though the notion seems very simple and is well known, we will need some less common version of it to give an adequate formal meaning to the construct of generalization (Section 4.2)—a main one in CM. To this end we will introduce the notion of *subvariant* type exactly dual to tuple type, and build two equivalent specifications for it. One goes along the common lines of viewing the variant type (subvariant type, I) and the other (II) is convenient for explaining generalization.

2.3.3. Subvariant type, I

A family of mappings, $g_i : Y_i \to X$, $i = 1, \ldots, n$, is called:

(i) *jointly covering or just covering*, if

$$[\texttt{cover}] \qquad \bigcup\{\mathrm{Rg}(g_i) | i = 1, \ldots, n\} = X;$$

(ii) *disjoint*, if all ranges $\mathrm{Rg}(g_i)$ are disjoint, that is,

$$[\texttt{disj}] \qquad \mathrm{Rg}(g_i) \cap \mathrm{Rg}(g_j) = \emptyset \text{ for any } i \neq j.$$

If, in addition, (iii) all mappings g_i are (single-valued and) injective (see Fig. 3(b′)), then X will be (isomorphic to) a disjoint union of $\mathrm{Dom}(g_i)$'s and thus will be a subset of disjoint union of Y_i's. In this case X might be called a *subvariant* type since variant type refers to complete disjoint unions.

Note, the notion of subvariant type is dual to the notion of tuple type (that might be called *subproduct*) while the variant type is, in fact, a diagram operation dual to the diagram operation of Cartesian product (see below).

2.3.4. Subvariant type, II

A family of mappings $f_i : X \to Y_i$, $i = 1, \ldots, n$ (note the reverse of arrows' directions) is called:

(i) *jointly total or just total*, if

$$[\texttt{total}] \qquad \bigcup\{\mathrm{Dom}(f_i) | i = 1, \ldots, n\} = X;$$

(ii) *disjoint*, if $\mathrm{Dom}(f_i)$ are disjoint:

$$[\texttt{disj}] \qquad \mathrm{Dom}(f_i) \cap \mathrm{Dom}(f_j) = \emptyset \text{ for any } i \neq j.$$

If, in addition, (iii) all f_i are (single-valued and) injective (Fig. 3(b″)), then the cotuple-mapping $f = \rangle f_1, \ldots, f_n \langle$ into the disjoint union of Y_i's,

$$f = \rangle f_1, \ldots, f_n \langle : X \to Y_1 + \cdots + Y_n, x.f \stackrel{\mathrm{def}}{=} x.f_i \text{ for that unique } i \text{ with } x \in \mathrm{Dom}(f_i),$$

is total and injective so that X is isomorphic to a subset of disjoint union of Y_i's, that is, a subvariant type.

It is easy to see that the family of injective mappings $f_i : X \to Y_i$ is jointly total and disjoint iff the family of inverse mappings (necessarily injective too) $g_i : Y_i \to X$, $g_i = f_i^{-1}$, is jointly covering (co-total) and disjoint. Thus, definitions of Sections 2.3.3 and 2.3.4 are equivalent. However, in the latter form, duality between the tuple type (Section 2.3.1) and subvariant type is more transparent. What was a property of each of the mappings f_i in Section 2.3.1 has become the corresponding joint property of the entire family (f_i, $i = 1, \ldots, n$) in Section 2.3.4 while what was a joint property in Section 2.3.1 is now required for each of the mappings.

2.3.5. Operations of Cartesian product and coproduct

These well known type forming operations are actually diagram operations. Their arity shapes are specified in Fig. 3(a1) and (b1) respectively (derived items are grey and dashed). Each of the operations, having a family of sets as its input, produces out of it a new set with a family of mappings having certain properties. The latter are specified by the corresponding predicate markers so that arity shapes of the operations in question are sketches rather than graphs. We have already explained semantics of the markers used besides the marker [cocover]. Its meaning is as follows.

A family of mappings $f_i : X \to Y_i$, $i = 1, \ldots, n$, is called *co-covering* if for any n elements $y_i \in Y_i$, $i = 1, \ldots, n$, there exist an element $x \in X$ such that $x.f_i = y_i$ for every i. If, in addition, the family is jointly monic (separating) in the sense of (Section 2.3.1), then there is exactly one such x and thus X is isomorphic to the set of all tuples built over $Y_1, \ldots, Y_n$. In other words, X is a Cartesian product $Y_1 \times \cdots \times Y_n$ up to isomorphism. Thus, Cartesian product is an operation producing a set with a separating and cocovering family of mappings as shown in Fig. 3(a1). The coproduct operation is defined in a similar (actually, dual) way as specified in Fig. 3(b1) (both views to variant type, I and II, are included).

We will finish our *tour-de-types-via-arrows* with sketching the set or grouping type. In a popular example, each object of class *Convoy* contains a set of *Ship*-objects and we say that *Convoy* contains a component of set/grouping type over class *Ship*.

2.3.6. Set/grouping type

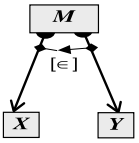
Instead of saying that a set X consists of subsets of another set Y, one can equivalently say that there is a binary relation $M \subseteq X \times Y$ which satisfies the following $\in$-condition:

$$[\in] \qquad (\forall x, x' \in X) x \neq x' \text{ implies } \{y \in Y : M(x,y)\} \neq \{y \in Y : M(x',y)\}$$

Indeed, in such a case the function f_M into the powerset of Y,

$$f_M : X \to \mathbf{Pow}(Y),\ x.f_M \stackrel{\text{def}}{=} \{y \in Y : M(x,y)\},$$

is injective and hence elements of X can be considered as unique names for some subsets of Y, namely, those that belong to the image of f_M. In a certain context, elements of X can be identified with these subsets of Y so that X is a *subpowerset* of Y up to isomorphism.



As for syntax, the considerations above show that to specify a node as a collection of subsets, one can safely leave the node without any marking but label instead the corresponding span diagram by a marker (say, a directed arc) denoting the constraint $[\in]$, as shown on the picture above.

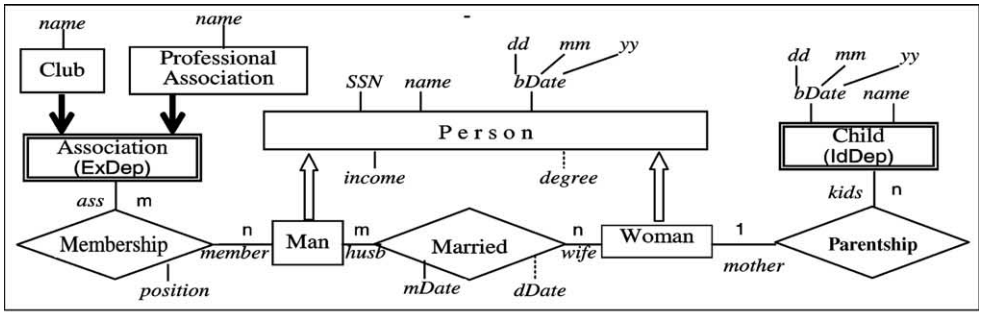
2.4. Conceptual modeling via sketches, I—static setting

In this section we will apply the framework developed so far to formal specifying a simple universe. We have designed it in such a way that its conceptual model could contain all major CM-constructs: entities and weak entities, relationships or associations in both forms—pure links and association classes, aggregations, *isA*-relationships in both forms—generalization and specialization, and, of course, attributes (including complex ones). We will present two conceptual models of the universe—in the ER and UML notations—and will try to explicate their precise formal meaning in the static sets-and-mappings framework. We will see what can, and what cannot, be done within the static sets semantics for sketches.

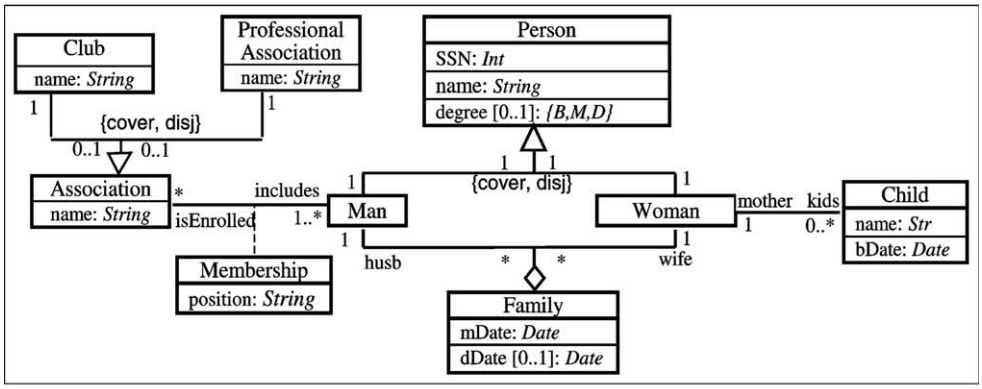
The universe to be modeled is as follows. Suppose, the user is interested in information about men, women and married pairs of some community for some period of time. Here "married pairs" mean pairs which are married or have been married during the period. In addition, the user wishes to track men's membership in professional societies and clubs, and parentship between children and women. In Fig. 4 three models of the universe in, respectively, ER, UML and SkeDM notations are presented.

2.4.1. ER- and UML-models

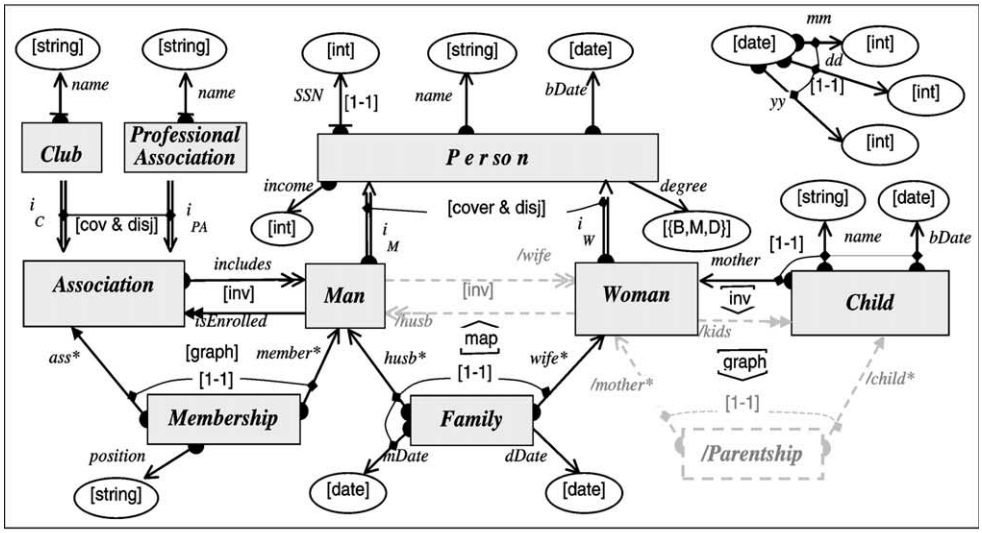
(1) The ER-diagram D' on the figure uses a conventional ER-notation. Semantics of nodes and attributes is hopefully clear from their names, *mDate*, *dDate* are dates of marriage and divorce, the latter is optional. The attribute *bDate* is of tuple type determined by its own attributes mm, dd, yy of type int (of integers). The classes Man and Woman are declared as specializations of the class Person (transparent *isA*-arrows) while class *Association* is declared as a generalization of



(a')



(a")



(b)

Fig. 4. Three models of a universe: ER, UML, Static Sketch. (a′) ER-diagram D′ (with an IFO-construct) (a″) UML-diagram D″; (b) static sketch, $S_{sta} = \mathbf{ske}[M^{ER}_{sta}(D') + M^{UML}_{sta}(D'')]$, where M^{*}_{sta} denotes a formal refinement of the intended semantic meaning of $*$-diagram in the static set framework.

classes *Club* and *profAssociation* (bold *isA*-arrows: here we use an element of the IFO metamodel [1]).

Note the double-framed node and label "ExDep" inside it for class *Association*. It means that the class is of *weak entity type* by the so called *existence dependency* [41]: the user is assumed to be interested only in *associations* in which the men from the universe are enrolled (so that deleting a man from the universe may enforce deleting an *association*). Also, we do not exclude the situation when some objects of classes *Club* and *profAssociation* do not occur into the class *Association* and thus bold *isA*-arrows are partially defined.

The class *Child* is also of weak entity type by the so called *identification dependency* [41]: each *child* is identified by her/his mother, birthdate and name.

(2) A slightly different presentation of the universe is given by the UML-diagram D''. There, *Married*-relationship is considered in a more solid setting as an *aggregation* of man in the role of husband and woman in the role of wife: the diamond adornment on the associations from *Man* and *Woman* to *Family* declares the latter class to be an aggregation. Also, the domain of the optional attribute degree is specified, it is the set consisting of three values B, M, D.

Then, because *Membership*-relationship has its own attributes and thus is of its own interest, it is modelled by an association class. In contrast, *Parentship*-relationship has no attributes and references to/from and thus it's reasonable to model it just as an association link. As for the identification dependency of class *Child*, it seems there are no direct graphical means in the UML to express it and thus it has to be done by a separate textual declaration attached to the class (we omit it in the diagram).

Finally, the generalization and specialization relationships between classes are both denoted by arrows with a hollow triangle head—the UML does not distinguish between specialization and generalization. Note also the constraints {cover} and {disj}, and multiplicities too, declared for these generalizations. [12]

2.4.2. Sketch model

Neither the ER-diagram nor the UML-diagram capture that delicate aspect of the situation that if a married pair $M = (w, m)$ got divorced and then w and m got married again, we may well wish to consider the same pair (w, m) as a new family M' distinct from M. In such a case, a proper identifier (key) for the class (relation) *Family* should be the triple (*wife*, *husb*, *mDate*) rather than the pair (*wife*, *husb*). While it is not easy to express such a constraint in ER and UML notational patterns directly, it can be quite naturally done in the sketch model of the universe, $\mathsf{S}_{\mathrm{sta}}$, presented in Fig. 4(c). Index "sta" is added to distinguish this static set model from a richer dynamic model we will build in Section 4.3.

In fact, sketch $\mathsf{S}_{\mathrm{sta}}$ is a blueprint of (the first draft of) formal semantics underlying the situation presented in the diagrams D' and D'' rather than a high-level model for which formal semantics should be recovered. Indeed, a first outline of the universe in formal terms would be as follows.

(1) We have a few abstract sets of main objects of interest, they are presented in Fig. 4(c) by grey rectangles, and a few mappings between these sets, they are denoted by arrows between rectangles. Sets *Man* and *Woman*, and *Club* and *profAssociation*, are subsets of, respectively, the

[12] Here we violate the actual UML syntax rules, where multiplicities for generalization edges are always set to be 1–1.

sets *Person* and *Association*, thus there are four inclusion mappings in the sketch (we remind that double-body of an arrow is actually a predicate marker declaring the arrow to be an inclusion). Two properties are declared for the both pairs of inclusion arrows: their images are disjoint and cover the target sets—note the markers [cover] and [disj] hung on the corresponding diagrams. [13] Note also that inclusions i_C and i_{PS} are partially defined.

Properties of mappings between sets are visualized by arrow's tails and heads specified in Fig. 1. For example, mappings *ass**, *member**, *husb** and *wife** are single-valued while *isEnrolled*, *includes*, *husb* and *wife* are multi-valued; also, the triangle heads of the arrows *ass** and */isEnrolled* declare the corresponding mappings to be covers.

(2) Objects of set *Family* are uniquely identified by triples of values $(wife^*, husb^*, mDate)$ as shown by marker [1–1] hung on the triple of the arrows so that they can be considered as tuples up to isomorphism (Section 2.3.1). Then, by applying the diagram operation $[\mathtt{map}\rangle$ to the span (see Fig. 2(c), we first take the binary *Man* × *Woman*-projection of the ternary relation Family and then apply $[\mathtt{map}\rangle$ to it), we get a pair of mutually inverse arrows (*/wife*, */husb*).

An "inverse" picture we have for the */Parentship*-fragment where the mapping mother is basic while the */Parentship*-relation together with projections (*/mother**, */child**) are derived by applying the operation $[\mathtt{graph}\rangle$, inverse to the operation $[\mathtt{map}\rangle$, to the arrow mother. The arrow */kids* is derived by operation $[\mathtt{inv}\rangle$ applied to *mother*. Note also the predicate label [1–1] hung on the triple (*mother*, *name*, *bDate*) which appears to be an adequate formalization of the identification dependency declared for the class *Child* in the ER-diagram.

Now consider the *Membership*-fragment of the sketch. It presents a bit different treatment of associations than those explicated in *Family*- and *Parentship*-fragments. Here the both constructs, pair (*includes*, *isEnrolled*) of inverse mappings and their graph $(Membership, ass^*, member^*)$, are considered to be basic items. Of course, they are mutually dependant and the marker [graph] is declared for the entire diagram. It means that

$$(Membership, ass^*, member^*) = [\mathtt{graph}\rangle(includes),$$

or, equivalently,

$$(Membership, ass^*, member^*) = [\mathtt{graph}\rangle(isEnrolled),$$

or, equivalently, [14]

$$(includes, isEnrolled) = [\mathtt{map}\rangle(Membership, ass^*, member^*).$$

Compare now the fragments of the sketch $\mathsf{S}_{\mathrm{sta}}$ adjoint to nodes *Membership* and *Parentship*: it appears to be a static approximation to the UML's distinction between association classes and pure association-links.

(3) Besides "rectangle" sets mentioned above (we have called their elements objects), we have four sets of different nature, which are denoted on the sketch by ovals: [int], [string], [date] and [{B, M, D}]. Elements of these sets are called *values* and the very sets are, correspondingly, *value domains*.

[13] Consisting of inclusion arrows together with their source and target nodes.

[14] We have used a linear string notation for writing diagram operations, it is of course a syntactic sugar which precise meaning can be easily recovered from arity shapes of the operations involved.

A fundamental distinction between object classes (sets of objects) and value domains (sets of values) is that extensions of the former are changeable and should be stored in the database while extensions of the latter are computable rather than storable. [15] To remind about this distinction, object class nodes are filled while value domains are transparent.

This fundamental distinction gets a quite natural expression in the sketch framework. Indeed, for the sketch approach, [int] and [{B, M, D}] are predicate labels (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, and if a node is marked by [{B, M, D}], its intended semantics is the set consisting of three elements B, M, D. In fact, we can consider value domains as 0-ary diagram operations producing nodes of known extent from empty input diagrams. Then, in accordance with our agreement on denoting derived items, we should depict value domains by blank rectangles with dashed grey frames and put their names (like, say, '*/Number*' for the domain of integers and '*/Degree*' for the domain [{B, M, D}]) in grey font too. However, two notational tips seem to be reasonable here.

First. Since the intended semantics of value domain nodes is fixed and remains the same for all schemas independently on their names, we adopt the convention of using the very operations markers instead of node's names. So, object class nodes are labeled by names (for example, *Person*) which do not bear any formal semantic meaning (but of course informally meaningful). In contrast, value domain nodes are labelled by markers (for example, [int]) specifying their intended semantics (while their names, say, *Number*, become redundant and can be omitted).

Second. In conceptual models, value domains play a special role of attributes' targets. Because of it, and following a notational tradition in CM of denoting "printable classes" by specially framed nodes, we have also visualized them more suggestively by oval frames. So, the oval shape is just a visualization means, syntactic sugar, rather than a predicate marker hung on nodes. Below, in Section 3.1.5 part (1), we will discover deeper semantic differences between object classes and value domains and then oval shape will become a predicate marker (of single-node arity) having a special formal meaning.

Finally, attributes of classes are nothing but mappings (arrows) from classes to value domains. Mandatory attributes are totally defined mappings (the arrows have bold tails) while optional attributes are partially defined. Note also the monic marker (Fig. 1) on the key attributes: *SSN* of *Person*, *name* of *Club* and *name* of *profAssociation*.

2.4.3. Discussion

We have considered a universe of discourse semi-formally specified by ER- and UML-diagrams in Fig. 4(a′) and (a″). Then we described the universe in abstract terms of ordinary (static) sets and mappings, and formally specified it by sketch S_{sta} in Fig. 4(b). What is fundamentally important with this sketch specification, is that nothing else is specified in S_{sta} apart its intended semantic meaning—in contrast to diagrams D', D'', S_{sta}'s semantics is explicit, simple and unambiguous. However, one may feel that this semantic picture is poorer than that embodied into the intuition behind the diagrams and some of its aspects are not captured by S_{sta}.

[15] As for the [{B, M, D}]-domain, its extension has to be stored but, importantly, together with the database schema rather than with the database extension as such.

First of all, the notion of object class, even on the level of data structures without behavioral aspects, is felt to be richer than being just a set, and specialization relationship between object classes, even w.r.t. data inheritance only, is richer than merely inclusion of sets. Also, the evident difference between changeable attributes like *income* or *degree* and *frozen* ones like *bDate* is not explicated, and, evidently, cannot be explicated in the static set framework. Also, it seems there is a deeper ontological difference between values and objects themselves (see, e.g., a discussion in [3]), hence, between object classes and value domains, beyond the difference of being computable–storable we have explicated so far.

Then, it is felt intuitively that there is an ontological difference between *Family*-relationships over classes *Man* and *Woman* and *Membership*-relationships over *Association* and *Man*. And indeed, in the UML diagram these relationships are presented by different modeling constructs of, respectively, aggregation and association class while in the sketch $\mathsf{S}_{\mathrm{sta}}$ they are the same. Somewhat similar question is whether the content of declaring the class *Association* a weak entity type by existence dependency is exhausted by totality of the arrow *includes*. Also, in the IFO-model and some other data metamodels, the principle distinction between specialization and generalization is stressed while in the sketch $\mathsf{S}_{\mathrm{sta}}$ they are presented in almost the same way: the only difference is that inclusions of subclasses into *Person* are total while inclusions into *Association* are partial. It seems there should be a deeper formal distinction between the two constructs.

So, sketch $\mathsf{S}_{\mathrm{sta}}$ is a precise specification with unambiguous formal semantics but the very semantics seems to be poorer than the intuition underlying the ER- and UML-models of the universe. In Section 4 we will see how to explicate this intuition, particularly, its deeper aspects mentioned above, in a much more adequate way and revise our example in a much richer formal semantic framework. But before that, some preliminary work on building the framework must be done, it is the goal of the next section.

3. Variable set semantics — . . .which are essentially changeable. . .

In the sketch paradigm for CM, the world consists of sets and mappings subjected to certain constraints. However, this is only the first main step of the approach. The second one is that these sets and mappings are *variable*, that is, changeable in time. So, nodes and arrows of a sketch must be interpreted dynamically.

3.1. Motivating example and main definition

3.1.1. Example

Let D be a conceptual model (diagram) of some universe, and one of the nodes is named '*Person*'. The intended semantics of this node is not a concrete static set of persons but rather a concept of *Person* relevant to the universe. That is, for one moment t there is one set of persons, $[\![Person]\!]^t$; its elements are persons as they were at the moment t. For another moment u, there is another set, $[\![Person]\!]^u$; its elements are persons as they were at the moment u; and so on. A bit more accurately, elements of set $[\![Person]\!]^t$ are considered to be *states* of persons (at the moment t) rather than persons themselves and, generally speaking, sets $[\![Person]\!]^t$ and $[\![Person]\!]^u$ may be disjoint. However, these sets are not independent of each other: they are interrelated by identifying different elements,

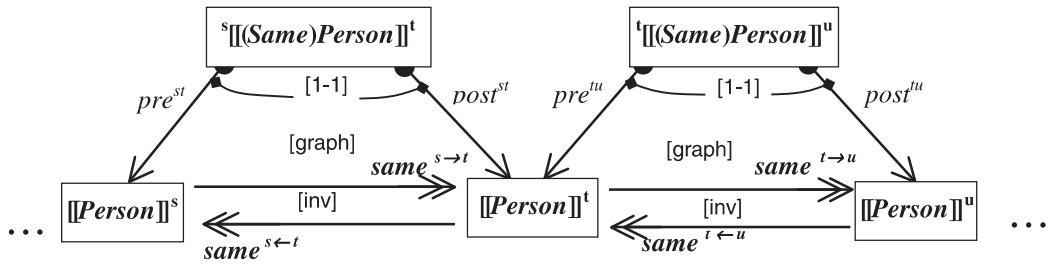


Fig. 5. Semantic of node *Person*, $s \leqslant t \leqslant u$ are time moments.

say, $p' \in [\![Person]\!]^t$ and $p'' \in [\![Person]\!]^u$, as different states of the same person, p. In this case we write $p' = p^t$ and $p'' = p^u$.

What is p? In the ER-paradigm a suitable concept is that of entity, in the OO-paradigm p would be called an object identity (o-id). However, usually these notion are not formally explicated and remains intuitive. [16] In contrast, we try to avoid scholastic considerations and take the constructive ontology view point that "to be" means to be observed, and so object identity is the observable identity. That is, if there are some verified reasons (a constructive proof as a mathematician would say) to consider p' and p'' to be (the two different states of) the same entity, this should be declared in an explicit form. Such a declaration, irrespectively to the ways of how it could be obtained, can be expressed by a family of binary inter-state relations

$$^t[\![SamePerson]\!]^u \text{ or just } ^t[\![Person]\!]^u \subseteq [\![Person]\!]^t \times [\![Person]\!]^u,\ t < u \text{ are time moments,}$$

so that $p' \in [\![Person]\!]^t$ and $p'' \in [\![Person]\!]^u$ have to be considered as different states of the same entity iff $(p', p'') \in {}^t[\![Person]\!]^u$.

Correspondingly, if for given t and $p^t = p' \in [\![Person]\!]^t$, for some later moment u no states p'' are observed in $[\![Person]\!]^u$ such that $(p', p'') \in {}^t[\![Person]\!]^u$ then the object p has to be considered removed from the universe during its evolution from t to u. Dually, if for some earlier moment s no states $'p$ are observed in $[\![Person]\!]^s$ such that $('p, p') \in {}^s[\![Person]\!]^u$, then the object p has to be considered appeared in the universe during the evolution from s to t.

Thus, a class is not a set but a chain of sets interconnected by binary relations between them, which trace the identity of the class' objects. This description is graphically specified by the sketch in Fig. 5 below. In this sketch, *Same*-nodes are binary relations with *pre*- and *post*-projection arrows (note the markers [1–1]). Each such a relations can be considered as the graph of a pair of mutually inverse *same*-mappings (note markers [graph] and [inv]). In general, *same*-mappings are partially defined and multi-valued.

Multiplicities of the interstate relations (*same*-associations) form a very important parameter of the picture and must be specified. It is the more so important since these multiplicities govern the ontology of the class' objects as we will see now.

[16] We remind that we talk about formalization without any references to computer related things like memory allocation, surrogates etc.

3.1.2. One–one same-relations

The most regular case is when multiplicities of all *same*-associations are 0..1, that is, interstate relations are of one–one type. Then object identity for any pair $t \leqslant u$ is set by partially defined injections (1–1 functions)

$$same^{t\to u} : [\![Person]\!]^t \to [\![Person]\!]^u$$

and their inverses

$$same^{t\leftarrow u} : [\![Person]\!]^u \to [\![Person]\!]^t.$$

So, for any $p' \in [\![Person]\!]^t$ and $p'' \in [\![Person]\!]^u$, $(p', p'') \in {}^t[\![Person]\!]^u$, i.e., p' and p'' present different states of the same object (entity) p, iff $p'.same^{t\to u} = p''$ or, equivalently, $p' = same^{t\leftarrow u}.p''$. Correspondingly, the two special situations of deleting and inserting objects can be expressed, respectively, by conditions $p'.same^{t\to u}$ is undefined and $same^{s\leftarrow t}.p'$ is undefined, $s < t < u$.

In this way the object identity of a person, say, *John*, is represented by a trajectory in the space of observable states:

$$John = (John^{t_1}, John^{t_2}, John^{t_3}, \ldots)$$

where $John^{t_i} \in [\![Person]\!]^{t_i}, John^{t_{i+1}} = John^{t_i}.same^{t_i\to t_{i+1}}$ and $t = t_1$ is the first moment when some element from $[\![Person]\!]^t$ was identified as *John*, that is, the t_1-state of John.

Thus, an object or entity is a trajectory as above and the meta-physical absolute o-id is replaced by a constructive notion of observable trajectory in the space of observable states.

3.1.3. Merging and splitting objects

Though the case of 0..1 multiplicities for the same-associations is the most natural, the situation of non-1–1 interstate relations is not excluded. Consider, for example, possible semantics of a node *Company*. It is a chain of sets of companies but the latter may merge and split and so relations ${}^t[\![SameCompany]\!]^u$ may be of many-to-many type when one has

(Merging)

$$(c', c) \in {}^t[\![Company]\!]^u \text{ and } (c'', c) \in {}^t[\![Company]\!]^u \text{ for some } c', c'' \in [\![Company]\!]^t, c \in [\![Company]\!]^u$$

and

(Splitting)

$$(c, c') \in {}^t[\![Company]\!]^u, (c, c'') \in {}^t[\![Company]\!]^u \text{ for some } c \in [\![Company]\!]^t, c', c'' \in [\![Company]\!]^u$$

Thus, generally speaking, the OO paradigm as such does not exclude manipulations with object identities, and in the two classical papers, [8,31] explicit operations to manipulate o-id were also suggested.

3.1.4. Logical time

Object identity is a construct for building a logical model of a universe rather than writing its chronicle. Then moments s, t, u should be treated more generally as indexes of real world states, and their precedence is logical rather than real. Correspondingly, the logical time is branching: it is ordered partially rather than linearly.

Considerations above motivate the following formal definition.

3.1.5. Definition—variable set

Let $\mathscr{T} = (T, \leqslant)$ be a partially ordered set called logical *time*: T is a set of *moments* or *state indexes* and $\leqslant \subseteq T \times T$ is a partial order of *logical precedence*.

Let $\mathscr{O}$ be some predefined collection of *observable states* of objects. A $(\mathscr{T}, \mathscr{O})$-variable set is a family of sets

$$C = (\llbracket C \rrbracket^t | t \in T, \llbracket C \rrbracket^t \subseteq \mathscr{O}),$$

taken together with a family of binary *interstate relations*

$$({}^t\llbracket C \rrbracket^u | t \leqslant u, {}^t\llbracket C \rrbracket^u \subseteq \llbracket C \rrbracket^t \times \llbracket C \rrbracket^u),$$

such that the following *reflexivity* and *transitivity* conditions hold for any $t, u, s \in T$:

$$(\text{Refl})_t \qquad (c', c'') \in {}^t\llbracket C \rrbracket^t \Longleftrightarrow c' = c'' \quad \text{for any } c', c'' \in \llbracket C \rrbracket^t$$

$$(\text{Trans})_{s \leqslant t \leqslant u} \qquad {}^s\llbracket C \rrbracket^t \bowtie {}^t\llbracket C \rrbracket^u = {}^s\llbracket C \rrbracket^u,$$

where $\bowtie$ denotes the operation of relational composition, i.e., the ordinary relational join in which the joined columns are removed.

When the context $(\mathscr{T}, \mathscr{O})$ is fixed, we will call $(\mathscr{T}, \mathscr{O})$-variable sets simply variable sets, or *varsets*. C is the name of the varset in question and sets $\llbracket C \rrbracket^t$, $t \in \mathscr{T}$, are *states* of C. Elements of $\llbracket C \rrbracket^t$ are called *states* of C-objects (at the moment t) and relations ${}^t\llbracket C \rrbracket^u$ are called same-associations for the class C.

Binary interstate relations is a construct symmetric w.r.t. past and future. In order to respect the direction of time, the functional notation for them, $same_C^{t \to u}$: $\llbracket C \rrbracket^t \to \llbracket C \rrbracket^u$, would be more suggestive and sometimes more convenient. Here $same_C^{t \to u}$ is considered to be a partially defined multi-valued mapping. The inverse mapping will be denoted by $same_C^{t \leftarrow u}$. We will call these functions *identity* or *same*-mappings.

As soon as the notion of o-id is explicated, one obtains the possibility to vary parameters and set different object ontologies. For example, one may prohibit splitting objects by requiring ${}^t\llbracket C \rrbracket^u$ to be a relation of many-to-one type ($same_C^{t \to u}$ is single valued) for all $t \leqslant u$. Dually, the requirement to be one-to-many prohibits merging objects. The condition of totally defined identity mappings determines the ontology where objects cannot be destroyed.

Another resource of setting different object ontologies is to vary the model of logical time: there is a variety of branching time models developed in theoretical computer science. In fact, they give rise to a whole area of temporal reasoning well known in AI.

So, the proposed framework is very flexible and covers a diversity of ontologies in a uniform way. Exploring this diversity seems to be an interesting and useful research direction. In this paper we will consider only a simple special case when *same*-mappings are a priori postulated to be partially defined injections. However, it turned out that even in this special case many phenomena in semantic modeling can be given an adequate formal meaning. In particular, we will formalize the *objects vs. values* and *entities vs. relationships* dichotomies, and all the semantic modeling constructs mentioned in Section 1 too.

Also, we will not use any properties of logical time besides an order on moments. The reader may safely consider time linear so that a mapping from $\mathscr{T}$ into $\mathscr{O}$ can be thought of as a trajectory. Linearity is useful for intuitive understanding but out of this reason scope it will not play any

essential role in our considerations. So, when we say 'trajectory' a reader inclined to more general setting can think of it as a mapping o: $\mathscr{T} \to \mathcal{O}$.

3.2. Variable sets and their properties. Taxonomy of objects

As it was said above, the taxonomy of varsets is rich and determined by the rich taxonomy of binary relations: by imposing different conditions on *same*-relations for a class, one can specify different ontologies of C-objects. We will consider the following three basic ontologies.

3.2.1. Definition

Let C be an object class, that is, a varset.

(i) C-objects are called *static*, if for any time moments $t \leqslant u$ the set $[\![C]\!]^t \cap [\![C]\!]^u$ is not empty and ${}^t[\![C]\!]^u$ is its diagonal, that is, the object identity mapping $same_C^{t \to u}$ is the identity mapping of the set $[\![C]\!]^t \cap [\![C]\!]^u$. Then object trajectories are glued into points; examples are values like integers or strings.

(ii) C-objects are called *proper*, if all relations ${}^t[\![C]\!]^u$ are of one–one type, that is, the object identity is a (partially defined) one–one mapping. Examples are entities in the conventional sense like *persons* or *vehicles*.

(iii) C-objects are called *legacyless*, or *snapshots*, if all ${}^t[\![C]\!]^u$ are undefined. In other words, each object trajectory is defined only for a single time moment so that the object has neither a history nor future. Examples are relationships in the sense of the ER-model like ownerships between *persons* and *vehicles*.

Correspondingly, we will call the very varset C *static*, *proper* or *legacyless* in function of the type of C-objects. Note, declaring C static does not mean that extension of C is not changeable: *states* of objects are static while the number of objects may change.

3.2.2. Remark—values as static objects

Values are evidently static objects but at first glance it seems there can be static objects other than values. However, let us consider the issue in the context of databases and let C be a node in a database schema. If it is a priori known, or assumed, that states of C-objects do not change, then states of C-objects can be considered as a component of the database schema rather than of the database states themselves. The latter statement just constitutes C-objects as values. Thus, we may define values as static objects.

3.2.3. Remark—relationships as legacyless objects

Exemplifying legacyless objects by conventional relationships needs explanation. What was said above about legacyless ownership means, more accurately, the following. There is a certain view on *ownerships*, let us call it *R-view*, in which the user considers pairs $(p, v) \in [\![Person]\!]^t \times [\![Vehicle]\!]^t$, $t \in T$, to be temporally discrete. This means that the question of whether the same pair (p, v) in two different but related states, $t < u \in \mathscr{T}$,

$$(p, v) \in [\![Person]\!]^t \times [\![Vehicle]\!]^t \text{ and } (p, v) \in [\![Person]\!]^u \times [\![Vehicle]\!]^u,$$

represents the same *ownership* or another ownership of the same objects is not stated, or considered incorrect. An example of different *ownerships* of the same objects is when p sold his vehicle v but then purchased it again in different terms.

Of course, there is another view on the ownership, *E-view*, when the user wants to trace identity of *ownerships* through time. Then the question of identifying two pairs of the same objects is legitimate, moreover, mandatory, and we consider the ownership as an entity association rather than a legacyless relationship. In general, it should be stressed that predicates of being a snapshot, or a value, or a proper object, are determined by user's (observer's) needs or possibilities rather than intrinsic properties of objects.

3.2.4. Remark—relationships vs. values

In both cases, when a varset C is legacyless and when it is a value domain, trajectories of C-objects are points but note a principle difference. In the first case trajectories are defined only for a single moment, in fact, the very notion of trajectory is *degenerated*. In the second case we have trajectories but they are of special kind: they are *reduced* to points.

3.3. Variable mappings and their properties

In the semantics of variable sets, to any arrow in a sketch, say, $owns : Person \to Vehicle$, is assigned the configuration of sets and mappings shown in Fig. 6. A variety of conditions can be declared for that configuration thus providing a variety of *dynamic constraints* to the arrow *owns*.

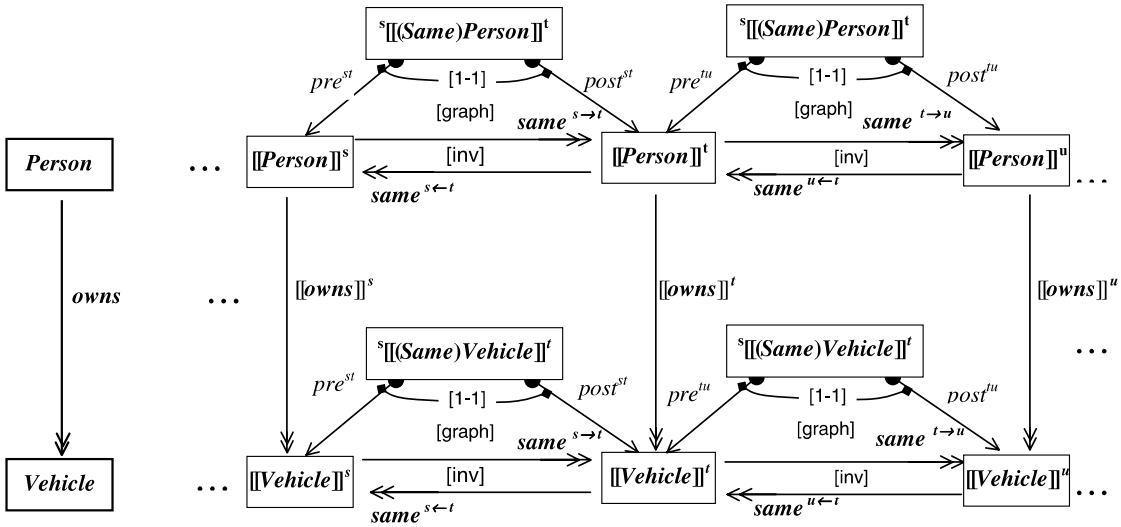


Fig. 6. Semantic of arrow $owns : Person \to Vehicle$. $s \leqslant t \leqslant u$ are time moments and binary *same*-relations are represented by pairs of their projection arrows, the horizontal arrows are the corresponding mappings.

3.3.1. Definition—variable mapping

Let C, D be variable sets as defined in Section 3.1.5. A *variable mapping* (or *varmap*) $f : C \to D$ is a family of mappings

$$f = (\llbracket f \rrbracket^t : \llbracket C \rrbracket^t \to \llbracket D \rrbracket^t | t \in T)$$

between ordinary sets—states of f's source and target.

From now on, the term 'arrow' will semantically mean a variable mapping. Because of the temporal dimension for variable mappings, the properties of arrows in the variable set semantics are much richer than in the static semantics. We will consider a few ones important for CM.

3.3.2. Definition—total and single-valued arrows

A variable mapping $f : C \to D$ is called (i) *total*, if $\mathrm{Dom}(\llbracket f \rrbracket^t) = \llbracket C \rrbracket^t$ for each moment t, and (ii) *single-valued* if $c.\llbracket f \rrbracket^t$ is a singleton for each t and any $c \in \llbracket C \rrbracket^t$.

In our example with arrow *owns*: *Person* → *Vehicle*, the mapping is total if every person *at any moment owns* some vehicles. The arrow is single-valued if each person may own not more than one vehicle *at any given time moment*. However, it does not prohibit owning a few vehicles during the entire life time of a *person-object*. Such a prohibition would be a quite another predicate (see definition 3.3.4 below).

3.3.3. Definition—locally and globally monic arrows

A possible dynamic constraint on a single-valued arrow $f : C \to D$ is to be one–one for each time moment t, that is, to be *monic locally*. For example, locally monic ownership means that at any given moment a vehicle may have only one owner.

Essentially another constraint is that a vehicle may have only one owner for the entire time period in question. Formally, a single-valued arrow $f : C \to D$ is called *globally monic/one–one/ injective* if for any $t < u \in T, c' \in \llbracket C \rrbracket^t, c'' \in \llbracket C \rrbracket^u$, one has

$$[1\text{–}1]^\infty \qquad (c'.\llbracket f \rrbracket^t, c''.\llbracket f \rrbracket^u) \in {}^t\llbracket D \rrbracket^u \text{ implies } (c', c'') \in {}^t\llbracket C \rrbracket^u$$

Globally monic mappings will be denoted by arrows with a double-arc tail ⊩.

3.3.4. Definition—static single-valued arrow

Dually to the definition above, a single-valued arrow $f : C \to D$ is called *globally comonic* if for any $t < u \in T, c' \in \llbracket C \rrbracket^t, c'' \in \llbracket C \rrbracket^u$ one has

$$[\mathtt{sta}_1] \qquad (c', c'') \in {}^t\llbracket C \rrbracket^u \text{ implies } (c'.\llbracket f \rrbracket^t, c''.\llbracket f \rrbracket^u) \in {}^t\llbracket D \rrbracket^u.$$

That is, in our example, a person cannot change the vehicle (s)he owns. We will use more suggestive names *static* or *frozen* for globally comonic arrows. Or else, such arrows and configurations built form them will be called *hard* while changeable arrows and configurations will be called *soft*.

For the general case of multi-valued mapping, the definition above needs some modification.

3.3.5. Definition—static multi-valued arrow

An arrow $f : C \to D$ is called *globally comonic* if for any $t < u \in T, c' \in \llbracket C \rrbracket^t, c'' \in \llbracket C \rrbracket^u$ one has

$$[\mathtt{sta}_m] \qquad (c', c'') \in {}^t\llbracket C \rrbracket^u \text{ implies } (c'.\llbracket f \rrbracket^t, c''.\llbracket f \rrbracket^u) \in \mathit{PowEqv}\ ({}^t\llbracket D \rrbracket^u).$$

where *PowEqv* (${}^t[\![D]\!]^u$) denotes the equivalence relation between subsets of $[\![D]\!]^t$ and $[\![D]\!]^u$ generated by ${}^t[\![D]\!]^u$ in the following way.

If $R \subset X \times Y$ is a binary relation then for any $A \subset X$, $B \subset Y$, $(A,B) \in PowEqv\ (R)$ iff, by definition, for any $a \in A$ there is some $b \in B$ s.t. $(a,b) \in R$ and for any $b \in B$ there is some $a \in A$ s.t. $(a,b) \in R$.

Other similar yet a bit different definitions of *PowEqv* (*R*) can be formulated and used in the definition $[\mathtt{sta}_m]$, it depends on the context of what one wants to consider as an unchangeable multi-valued mapping.

3.3.6. Static universe

Static arrows between static sets (see Section 3.2.1) constitute the static sub-universe of the dynamic universe of varsets. In particular, value domains are static sets and a function f into a value domain is an attribute of the class C. Then the property of being static means that for any object in C its f-attribute does not change. In other words, static attributes (e.g., *birthDate* or *name* when we assume that names do not change) are attributes of an object. In contrast, attributes which are not static are attributes of object states (e.g., *age* or *salary*).

3.3.7. External identifiers

The property of being both global monic and static, that is,

$$[1\text{–}1]^{\infty}\&[\mathtt{sta}_1] \qquad (c',c'') \in {}^t[\![C]\!]^u \Longleftrightarrow (c'.[\![f]\!]^t, c''.[\![f]\!]^u) \in {}^t[\![D]\!]^u$$

is remarkable. In this case, D-values of f can serve as globally unique and static representatives of C-objects, that is, be *external identifiers* of C-objects.

The static arrow conditions $[\mathtt{sta}_1], [\mathtt{sta}_m]$ say nothing about relation between the values of f at c at moments $t \leqslant u$ if $c'.same_C^{t\to u}$ is undefined, that is, object c is deleted from the universe during its evolution from t to u. Similarly, they say nothing if $c''.same_C^{t\leftarrow u}$ is undefined, that is, the object c was inserted into the universe during its evolution from t to u. These situations are considered below.

3.3.8. Definition—del- and ins-static arrows

(i) A static arrow $f : C \to D$ is called *del-static* if it is compatible with object deletions: for any $c \in [\![C]\!]^t$,

$[\mathtt{del\text{-}sta}_m]$ if $c.same_C^{t\to u}$ is undefined then $d.same_D^{t\to u}$ is also undefined for each $d \in c.[\![f]\!]^t$.

That is, if our ownership arrow *owns* : *Person* → *Vehicle* is del-static then the vehicles of a person are deleted as soon as the person is deleted.

(ii) A static arrow $f : C \to D$ is called *ins-static* if it is compatible with object insertions: for any $c \in [\![C]\!]^u$,

$[\mathtt{ins\text{-}sta}_m]$ if $same_C^{t\leftarrow u}.c$ is undefined then $same_D^{t\leftarrow u}.d$ is also undefined for each $d \in c.[\![f]\!]^u$.

That is, if arrow *owns*: *Person* → *Vehicle* is ins-static then all the vehicles of a newly inserted person must be also newly inserted.

(iii) A static arrow is called *del–ins-static* if it is both del-static and ins-static.

3.4. Diagram predicates over varset universes

All properties of arrow diagrams considered in Section 2 are immediately transferred into the varset framework by considering them locally, that is, by adding the quantifier "for any moment t". For example, a family of functions $(f_i : C \to D_i, i \in I)$ is called *locally separating/locally jointly monic/local key* if at any time moment $t \in T$ the family $(\llbracket f \rrbracket^t : \llbracket C \rrbracket^t \to \llbracket D_i \rrbracket^t, i \in I)$ is separating (see Section 2.3.1). Similarly, a family $(f_i : C_i \to D, i \in I)$ is called *locally disjoint* if for any $t \in T$ the family of mappings $(\llbracket f \rrbracket^t : \llbracket C_i \rrbracket^t \to \llbracket D \rrbracket^t, i \in I)$ is disjoint. However, these constraints have essentially different (stronger) global versions where dynamics of the varset world is involved much more essentially.

3.4.1. Definition—global identifiers

Let $(f_i : C \to D_i, i \in I)$ be a family of total single-valued mappings (an arrow source).

(i) The family is called *globally separating*, or *globally monic* (*one–one*), or else a *global key*, if for any $t < u \in T$ and $c \in \llbracket C \rrbracket^t, c' \in \llbracket C \rrbracket^u$ the following holds:

$[1\text{–}1]^\infty \quad (c', c'') \notin {}^t\llbracket C \rrbracket^u$ implies $(c'.\llbracket f_i \rrbracket^t, c''.\llbracket f_i \rrbracket^u) \notin {}^t\llbracket D \rrbracket^u$ for some $i \in I$.

Equivalently, $(c', c'') \in {}^t\llbracket C \rrbracket^u$ as soon as $(c'.\llbracket f_i \rrbracket^t, c''.\llbracket f_i \rrbracket^u) \in {}^t\llbracket D \rrbracket^u$ for all $i \in I$.

(ii) The family is called static, if all f_i are static.

(iii) The family is called a *proper identifier*, if it is both globally separating and static.

The arrow source's property to be both globally monic and static is remarkable. In this case, there is a global—through the entire time—one–one correspondence between C-objects and tuples $(d_i, i \in I)$ where each d_i is a D_i-object (that is, in the constructive ontology view, a trajectory). In particular, when all D_i's are value domains, C-objects are globally uniquely identified by tuples of D_i's values. Also, it is easy to see that a global key is always a local key but the converse is not true.

3.4.2. Example

Let C be a class of employees of some company. Suppose that each employee is identified by his name and a number to distinguish between persons with the same name. The following disciplines of managing identification are possible.

(a) If an employee, say, $(John, 5)$, leaves the company then other *employees* are re-enumerated to fill the gap: $(John, 6)$ becomes $(John, 5)$ and so on. Then the pair of attributes $(name, number)$ is locally a key but it is neither static nor global.

(b) If $(John, 5)$ leaves the company, then the identifier $(John, 5)$ becomes vacant and can be assigned to a new employee who joins the company. In this case the pair $(name, number)$ is a static key but it is not global.

(c) The same as (b) but the identifier $(John, 5)$ cannot be assigned to another employee even if the person $(John, 5)$ leaves the company. In this case the pair $(name, number)$ is a proper identifier.

(d) The same as (c) but employee are allowed to change their names, e.g., $(John, 5)$ may become $(Johann, 2)$. However, the previous identifier, $(John, 5)$, remains reserved. Then the pair $(name, number)$ is a global key but it is not static.

3.4.3. Definition—globally disjoint arrow families

A family of arrows $(f_i : C_i \to D, i \in I)$ is called *globally disjoint* if for any time moments $t \leqslant u$ and any elements $d' \in [\![D]\!]^t$, $d'' \in [\![D]\!]^u$,

$$((d', d'') \in {}^t[\![D]\!]^u \quad \text{and} \quad d' \in c'.[\![f_j]\!]^t \text{ for some } c' \in [\![C_j]\!]^t \quad \text{and} \quad d'' \in c''.[\![f_k]\!]^t \text{ for some } c'' \in [\![C_k]\!]^t) \text{ implies } j = k.$$

A family of subsets is called *globally disjoint* if the corresponding family of inclusion mappings is globally disjoint.

Note, any globally disjoint family is locally disjoint by reflexivity of *same*-relations but the converse is not true in general.

3.4.4. Example

Let a class *Person* has two subclasses *Engineer* and *Manager*. If an engineer cannot be simultaneously a manager but an engineer can become a manager (thus leaving the position of engineer), then the pair of inclusion mappings is disjoint locally but not globally.

4. Conceptual modeling and object identity — . . .and ontologically connected

The world is full of inter-dependencies between objects: objects may form aggregations of different types (e.g., a family aggregates the wife, the husband and children); may occur in other object (an engine is a part of car); may be specialization of other objects (an employee is a specific kind of person) and so on. All these relations between objects have a certain ontological nature with sometimes rather subtle peculiarities. For example, a car also aggregates the engine, the chassis, the four wheels but this aggregation is perceived intuitively different from the family aggregation mentioned above; in particular, we would hardly say "the wife is a part of family" (at least, in some common context). A major concern of conceptual modeling is just to specify these ontological phenomena in abstract and precise terms. Variable set semantics explicates the notion of object identity but what is to do with ontological relations between objects like those listed above?

Fortunately, the answer is an almost immediate consequence of the varset idea. It follows from the latter that, given an object class C, identity of C-objects is explicated by interstate relations ${}^t[\![C]\!]^u, t \leqslant u$, and thus, ontology of C-objects is determined by C-interstate relations. Then, a natural way to explicate that C-objects somehow ontologically depend on objects of other classes $D_1, D_2, \ldots, D_n$ is to state that interstate relations for C depend on those for D_i, that is, ${}^t[\![C]\!]^u$ can be somehow derived from ${}^t[\![D_i]\!]^u, i = 1, \ldots, n$, for each pair of moments $t \leqslant u$.

In this section we will see how it works for formal specifying well known constructs of conceptual modeling, such as aggregation, association, composition, generalization and a few novel constructs related to these (of course, the latter appear in applications but it seems were not explicitly specified in the literature). In naming these constructs we will try to follow the UML terminology and also the terminological tradition of the ER-model.

4.1. Formal semantics for aggregation relationships between object classes

4.1.1. Aggregation

4.1.1.1. Motivating example. Consider, again, a fragment of our main universe in Section 2.4, Fig. 4, concerning *Family*-objects (*Married*-pairs on the ER-diagram). We have finished our formal specification of the universe in Section 2.4 with sketch S_{sta} stating, in particular, that for any time moment t and for any two objects o', o'' of the class *Family* we have

$$\begin{aligned} o' = o'' \text{ in } [\![Family]\!]^t \text{ iff } \quad & o', [\![wife]\!]^t = o''.[\![wife]\!]^t \text{ in } [\![Person]\!]^t \\ & \& \; o'.[\![husb]\!]^t = o''.[\![husb]\!]^t \text{ in } [\![Person]\!]^t \\ & \& \; o'.[\![mDate]\!]^t = o''.[\![mDate]\!]^t \text{ in } [\![\texttt{date}]\!] \end{aligned} \tag{3}$$

(we have omitted star indexes near projection arrow names *wife* and *husb*). So, *Family*-objects are identified by pairs (*wife*, *husband*) (with *mDate*-qualification if necessary) and, if we assume that persons are uniquely identified by their names, we may say, for example, ''the family of Mary and John (married on mm/dd/yy)''. However, the equivalence (3) states a static identification of *Family*-objects while the world is changeable and later Mary may well become Miriam and John may become Johann. So, we need a dynamic identification of *Family*-objects as follows: for any moments $t \leqslant u$ and any object states $o' \in [\![Family]\!]^t, o'' \in [\![Family]\!]^u$,

$$\begin{aligned} (o', o'') \in {}^t[\![Family]\!]^u \text{ iff } \quad & (o', [\![wife]\!]^t, o''.[\![wife]\!]^u) \in {}^t[\![Person]\!]^u \\ & \& \; (o'.[\![husb]\!]^t, o''.[\![husb]\!]^u) \in {}^t[\![Person]\!]^u \\ & \& \; (o'.[\![mDate]\!]^t = o''.[\![mDate]\!]^u) \end{aligned} \tag{4}$$

and thus the triples (Mary, John, 05/12/95) and (Miriam, Johann, 05/12/95) refer to the same family since the pairs (Mary, Miriam) and (John, Johann) refer to the same persons. (Note, equivalence (3) is a particular case of (4) for $u = t$ because of reflexivity condition for interstate relations).

Equivalence (4) has to be considered as a *definition* of interstate relations for class *Family*. In fact, it specifies how to derive relations ${}^t[\![Family]\!]^u$ from relations ${}^t[\![Person]\!]^u$ having defined mappings *wife*, *husb*, *mDate* and we may rewrite it as follows:

$${}^t[\![Family]\!]^u \stackrel{\text{def}}{=} \{(o', o'') \in [\![Family]\!]^t \times [\![Family]\!]^u | \text{ right side of equivalence (4) holds}\}. \tag{5}$$

What we have just described is precisely specified by sketch S_3 in Fig. 7. The marker $[\texttt{ID}^{\times}]$ hung on the triple of arrows (*wife*, *husb*, *mDate*) declares it to be the $\times$-*identifier* or the (*primary*) $\times$-*key* to the class *Family* in the sense of the dynamic definition above. The index $\times$ points to a *conjunctive*, product-like, way of identification used in (4) and is essential since below we will introduce also disjunctive $+$-identification. The node *Family* is double-framed to emphasize that the interstate relations are derived (via the mappings occurring into the identifier). [17] The (variable) attribute *income* conventionally represents multiplicity of attributes and references to other classes the class

[17] Such a node represent a kind of weak entity class, and we use ERD's notation for weak entities though, of course, the meaning of the predicate ''weak'' here is quite different.

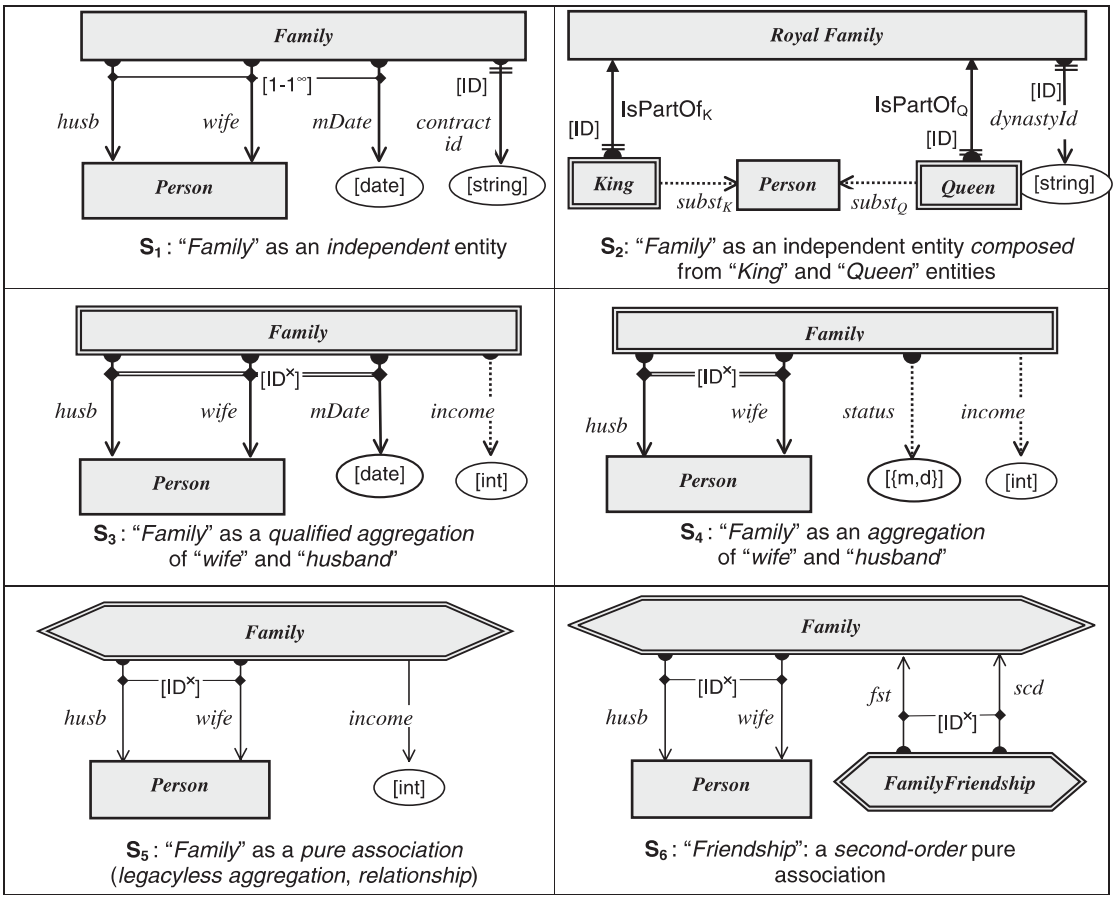


Fig. 7. Different conceptual views on the same universe.

Family may have; all they perform a *descriptive* function in contrast to the identifying function the triple (*wife*, *husb*, *mDate*) performs.

To clarify the meaning of the construction we have described, let us consider another, principally different, view on *Family*-objects, where they are seen from the pure juridical side and identified by the marriage contract as shown on sketch S_1 (Fig. 7, note the marker [ID] hung on arrow *contract-id*). The triple (*wife*, *husb*, *mDate*) is again declared to be globally one–one and static, and thus the equivalence (4) holds again. The crucial difference is however that in contrast to the view specified by sketch S_3 where (4) is the definition of ${}^{t}[\![Family]\!]^{u}$, for the view presented by sketch S_1 it is a constraint to the triple (*wife*, *husb*, *mDate*) (both sides of (4) are defined independently of each other and (4) sets a logical constraint they must satisfy). If the constraint is indeed satisfied, the triple (*wife*, *husb*, *mDate*) can be used for external identification of *Family*-objects but it has no ontological meaning. In the view specified by sketch S_1 references *wife*, *husb*, *mDate* perform descriptive rather than ontological function.

It seems that derivability of *Family*-object identity expressed in (5) formally explicates the intuition that a *Family*-object *consists* of a *Person*-object in the role of *wife* and *Person*-object in the

role of husb together with aggregated *mDate*-values. In the UML-terms, definition (5) presents the class *Family* as an *aggregation* of *Person*-objects (see [37] for UML-references). In contrast, in the view specified by sketch S_1, *Family*-objects are *independent entities*: their o-id do not depend on objects of other classes. The class *Family* is said to be of *independent entity* type and the node denoting it on sketch is single-framed.

4.1.1.2. Remark. Though in the view by sketch S_3 the o-id of the class *Family* is derived, its extension is not derivable and has to be specially stored: not every pair of *Person*-objects forms a *Family*-object.

4.1.1.3. Formal definition—aggregation of object classes. An object class (varset) C is called an *aggregation* of classes $D_1, \ldots, D_n, n \geqslant 2$, if there are total single-valued arrows (variable mappings) $f_i : C \to D_i$, $i = 1, \ldots, n$, such that C's interstate relations are derived from those of classes D_i's as follows: for any time moments $t \leqslant u$ and object states $c' \in [\![C]\!]^t, c'' \in [\![C]\!]^u$

$$(c', c'') \in {}^t[\![C]\!]^u \text{ iff, by definition, } (c'.[\![f_i]\!]^t, c''.[\![f_i]\!]^u) \in {}^t[\![D_i]\!]^u \text{ for all } i = 1, \ldots, n \tag{6}$$

In such a case the family of arrows $(f_i, i = 1, \ldots, n)$ is called the *×-identifier* of class C, or else the *primary* ×-key to the class C; the way of identification is called *conjunctive*. Below we will enter a dual type of identification—*disjunctive* identification by +-identifiers (Section 4.2.2.3).

Aggregation is called *pure* (*strong*) if all D_i's are object classes. Aggregation is called *qualified* (*structurally weak*, *s-weak*) if some of D_i's are value domains and, hence, some of f_i's are attributes rather than references to other classes.

It follows immediately from the definition that for any given moment t, the family $([\![f_i]\!]^t, i = 1, \ldots, n)$ is jointly one–one. To see this it is sufficient to set $u = t$ and apply reflexivity of the *same*-relations. Also, by the very definitions, any arrow f_i is trivially static and the entire family is trivially globally one–one.

4.1.1.4. Examples. One example was already considered and specified by sketch S_3 in Fig. 7. Aggregation there is qualified by values of *mDate*-attribute. To obtain a *pure aggregation* of *Person*-objects, we have to suppose that *Family*-objects are identified on the base of only two attributes: *wife* and *husb*. That is, for any two moments $t \leqslant u$ and objects $o' \in [\![Family]\!]^t$, $o'' \in [\![Family]\!]^u$,

$$\begin{aligned}(o'o'') \in {}^t[\![Family]\!]^u \text{ iff } \quad & (o'.[\![wife]\!]^t, o''.[\![wife]\!]^u) \in {}^t[\![Person]\!]^u \\ & \& (o'.[\![husb]\!]^t, o''.[\![husb]\!]^u) \in {}^t[\![Person]\!]^u\end{aligned} \tag{7}$$

In fact, in this view *Family*-objects have an attribute *status* which takes one of the two values—*married* or *divorced*, and this attribute is a descriptor out of the identifier. Thus, the view we are considering is specified by the sketch S_4 in Fig. 7.

The formal model we have entered also recovers a few variants of aggregation which are not widely known to the community, at least, in a explicit form. Particularly, we may consider situations when the key to an aggregation class has some defects so that definition 4.1.1.3 must be modified to compensate them. Details can be found in [9] where such deficient aggregations are called *temporally weak* (*t-weak*).

4.1.2. UML's association or ER's relationships

Consider a situation when marriage dates are not accessible to the user. Moreover, let us assume that now the user does not at all concern whether a given pair $(Mary,\ John) \in [\![Family]\!]^u$ represents the same family as $(Mary, John) \in [\![Family]\!]^t$, $t \leqslant u$, or a new one. In other words, the user prefers to consider families as instantaneous objects without a history or future and, in fact, is interested only in moment snapshots of which pairs are currently married. Then the pair (*wife*, *husb*) is to be considered as the local identifier of the class *Family* while its global uniqueness is not interesting.

A formal definition is as follows: for any two moments $t \leqslant u$ and objects $o' \in [\![Family]\!]^t$, $o'' \in [\![Family]\!]^u$,

$$\begin{aligned}(o', o'') \in {}^t[\![Family]\!]^u \text{ iff } \quad & (o'.[\![wife]\!]^t, o''.[\![wife]\!]^u) \in {}^t[\![Person]\!]^u \\ & \& \ (o'.[\![husb]\!]^t, o''.[\![husb]\!]^u) \in {}^t[\![Person]\!]^u \\ & \& \ u = t\end{aligned} \tag{8}$$

In this definition, in contrast to similar definition (4), the third row does not provide any additional information about interstate identification but, instead, removes the very necessity in such information. So, the *Family*-class appears as a legacyless varset in the sense of 3.2.1 and is called *legacyless aggregation* of *Person* and *Woman*. Correspondingly, a better name for the class would be a present participle like *Married* rather than a noun. The class is specified by the sketch S_5 in Fig. 7, where the rectangle–diamond hybrid node denotes a legacyless varset, that is, a legacyless identification of *Family*-objects. So, the special frame of the node is a marker of the entire span diagram rather than of the node itself, and we have chosen a diamond component in the notation to remind about the classical ER-notation for relationships.

4.1.3. Second-order association

In the view S_6, an additional second-order legacyless aggregation class is specified: a *Friendship* relation between *Family*-pairs. Its object identity is given by the definition:

$$(o', o'') \in {}^t[\![Friendship]\!]^u \text{ iff } (o'.fst, o''.snd) \in {}^t[\![Family]\!]^u \tag{9}$$

so that o-id of this class is derived without any additional conditions. It means that the class *Friendship* is a strong aggregation but of course is legacyless too. This example shows that identity of the aggregation class' objects is not "stronger" than components' identities (see [12] for details).

4.1.4. Composition

A special case of aggregation is represented by sketch S_2 in Fig. 7. It is an "aggressive" aggregation when component objects loss their own identity and begin to be identified by the aggregation object. That is, a person in the role of King (Queen) is identified by the reference to the corresponding *Royal Family*-object and we talk about "the King of the Plantagenet Royal Family" (quite similarly to, say, talking about "the chassis of the car # 555"). In a sense, *King*-objects and *Queen*-objects appear as parts of *Royal Family*-objects.

A formal specification of this context is presented by sketch S_2 in Fig. 7. The arrows serving as keys to component classes are named *isPartOf*; names of their inverse arrows can be taken to be the same as their target classes (*King*, *Queen*) and so are omitted. Clearly, these arrows are hard (non-changeable) as any key arrows and inverse to them and thus with these arrows the classes

Royal Family, *King* and *Queen* are connected into a hard configuration. In this configuration, objects of the components classes, *King* and *Queen*, are, in fact, positions (roles) for which real world objects from class *Person* are substituted. This substitution is specified by arrows

$$subst_K : King \to Person \quad \text{and} \quad subst_Q : Queen \to Person$$

which are single-valued, partially defined (positions may be vacant) and, in general, soft (changeable). Note, we consider the very positions as mandatory components of *Royal Family* and thus arrows *isPartOf* are covers, but we admit non-fulfilled positions and thus *subst*-arrows are partial.

It seems that the kind of aggregation between objects just considered captures some aspects of the intuition underlying UML's construct of composition. So, the class *Royal Family* appears as a composition of classes *King* and *Queen*. We do not assert that the formal construction we have described provides a complete model for UML's composition. No doubts that behavioral aspects, which are a priori beyond our framework, are really essential for semantics of composition and without them any model will be incomplete. However, we do assert that object identification is a part of composition's semantics, and our construction gives a formal meaning for this part.

4.2. Formal semantics for specialization and generalization relationships between object classes

The relationships in the title are fundamental constructs of CM. Their idea itself is clear but there are some delicate points still not presented in the literature. Usually both constructs are based on the notion of *isA*-arrow between object classes but some authors distinguish between specialization and generalization *isA*-arrows (see [1] for the most careful treatment we know).

However, in the light of formal semantics we will build for these constructs in this section, it will be seen that specialization is indeed based on the notion of *isA*-arrow while in generalization a different type of arrows is used.

The counterpart of *isA*-construct on the level of implementation is the concept of inheritance where two aspects are distinguished: data (structure) inheritance and procedure (behavior) inheritance. The latter is a main concern of OO-programming and goes out the scope of the paper. In the context of conceptual modeling, we address only to data inheritance where the following issues were identified in the literature [1,37,39]:

- distinction between specialization and generalization,
- update propagation along *isA*-networks,
- correctness conditions for *isA*-networks.

We consider the first two in the present section and the last one is addressed below in Section 4.3.

4.2.1. Specialization and isA-relationship

4.2.1.1. Motivating consideration. Suppose that our conceptual schema contains an object class *Employee* and another class *Engineer* considered as specialization of *Employee*, that is, *Engineer*-objects are considered *Employee*-objects having some special additional properties. This is also

expressed by declaring an *isA*-relationship (arrow) from *Engineer* to *Employee*: "every engineer is an employee".

What is a formal meaning of this declaration? First of all, it means that for any moment t

$$[\![Engineer]\!]^t \subseteq [\![Employee]\!]^t \tag{10}$$

but this is only a poor static projection of the phenomenon, which does not take into account the dynamic nature of object identity. Indeed, the actual meaning of the declaration "every engineer is an employee" seems have to include, besides inclusions (10), the following: if two states of Engineer-objects, say, (John,5) and (Johann,2) are identified as representing the same employee, then they automatically represent the same engineer as well. Formally, for any two states, t and u, such that the state t logically precedes the state u, the interstate relation for the class *Engineer*, ${}^t[\![Engineer]\!]^u$, is the restriction of the interstate relation for the class *Employee*, ${}^t[\![Employee]\!]^u$:

$${}^t[\![Engineer]\!]^u = {}^t[\![Employee]\!]^u \cap ([\![Engineer]\!]^t \times [\![Engineer]\!]^u). \tag{11}$$

Note, this equality implies inclusion ${}^t[\![Engineer]\!]^u \subseteq {}^t[\![Employee]\!]^u$ but is stronger. Thus, as soon as the t-extension and the u-extension of the node *Engineer* are given, the corresponding interstate relation is derived from the *Employee*-interstate relation and so object identity of engineers is derived from that of employees. Such a treatment seems to reveal the ontological sense of the declaration "any engineer is an employee".

It is easy to see that the condition above can be equivalently rewritten in the form:

$${}^t[\![Engineer]\!]^u \stackrel{\text{def}}{=} \{(o', o'') \in [\![Engineer]\!]^t \times [\![Engineer]\!]^u | (o'.[\![isA]\!]^t, o''.[\![isA]\!]^u) \in {}^t[\![Employee]\!]^u\} \tag{12}$$

where *isA* denotes the inclusion mapping from the class of engineers to the class of employees. This condition is nothing but a particular case of similar condition (6) and so means that the arrow *isA* is the identifier of the class *Engineer*.

Thus, declaring class *Engineer* to be a specialization of the class *Employee* amounts to the following three points:

- there is a variable mapping from the varset *Engineer* to the varset *Employee*,
- this mapping is inclusion of $[\![Engineer]\!]^t$ into $[\![Employee]\!]^t$ for any time moment t,
- this mapping is declared to be the identifier of the class *Engineer*.

4.2.1.2. Formal definition—specialization (isA-)relationship between object classes. A class (varset) C is a *specialization* of a class D if there is an arrow (variable mapping) $f : C \to D$ such that

(i) $[\![f]\!]^t$ is a totally defined inclusion of $[\![C]\!]^t$ into $[\![D]\!]^t$ for any time moment t, and
(ii) f is declared to be the identifier of the class C: for any $t \leqslant u \in \mathscr{T}, c' \in [\![C]\!]^t, c'' \in [\![C]\!]^u$,

$$(c', c'') \in {}^t[\![C]\!]^u \text{ iff, by definition } (c'.[\![f]\!]^t, c''.[\![f]\!]^u) \in {}^t[\![D]\!]^u. \tag{13}$$

In such a case we call f an *isA*-arrow which states an *isA-relationship* between classes C and D, C is called the *subclass* and D the *superclass*.

4.2.1.3. Notation. Following a tradition, an inclusion arrow declared to be the identifier of the source class will be denoted by a double-body arrow. It is always assumed to be totally defined and we will omit the tail marker of totality.

4.2.1.4. Example. The role of condition (ii) in the definition of *isA*-arrows can be seen from comparison of sketches S_1, S_2 in Fig. 8 specifying two different ways of object identification for class *Child*.

Semantics of the left sketch is clear. The class *Child* is declared to be in a *isA*-relationship to the class *Person* and hence a *child* is identified by her/his *SSN*. Notice inequality constraints for two diagrams in the sketch: they express the basic for the universe fact that a *child* cannot be her own parent.

In the view presented by the sketch S_2, a *Child*-object is identified by the triple (*mother*, *name*, *bDate*′). Then, though the arrow *person* : *Child* → *Person* is globally one–one and static, it is not an *isA*-arrow. In relational terms, the attribute *person* of the table *Child* is a foreign key referencing to the table *Person*. The latter also has the attribute *bDate* and so, an integrity constraint must be declared: for any *child*-tuple, its attribute *bDate*′ must coincide with the attribute *bDate* of the tuple referenced by the foreign key (note marker [=] declaring commutativity: *person* ⋗ *bDate* = *bDate*′). In contrast, in sketch S_1 the attribute *bDate* of a *Child*-object is defined to be the *bDate* of the corresponding person.

4.2.1.5. Specialization diagrams. In practice, specialization often occurs in conceptual models in the form of a complex *isA*-network rather than a single *isA*-arrow or a set of disconnected *isA*-arrows. A typical situation is the top-down design of class schemas. For example, one begins with a class *Employee*, then considers two its subclasses, *Engineer* and *Manager* and, if it is needed, takes also their intersection, class MgrEngineer (the result is presented in the top row of Fig. 9). Additional requirements of disjointness of the subclasses, or their jointly covering property, or both, can be also declared for the entire specialization diagram (the second row of Fig. 9).

An important property often attributed to specialization (e.g., [1]) is that transitions of objects between subclasses are allowed. In our example, engineers may become managers and vice versa. Note however that other situations are also possible, for example, that one described in the third row in Fig. 9. Because of a special condition of object transition between subclasses and the

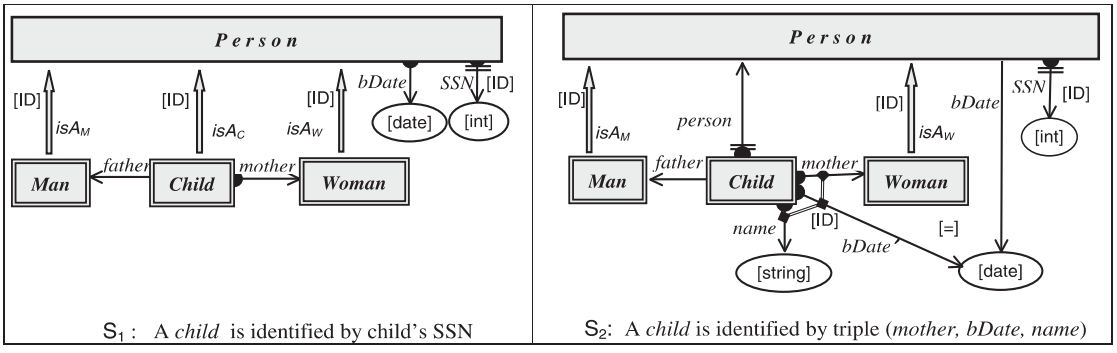


Fig. 8. The nature of *isA*-arrows.

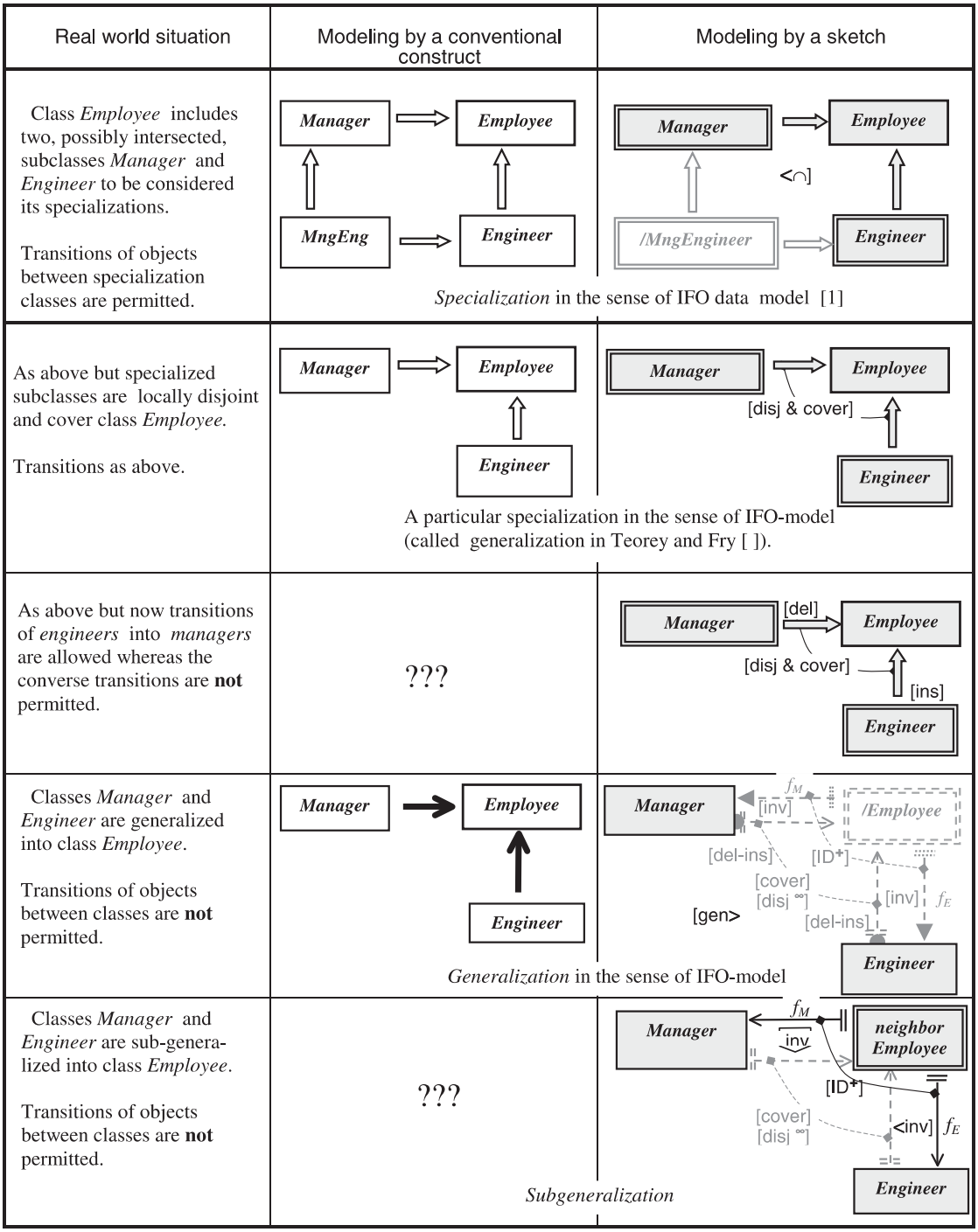


Fig. 9. Diversity of *isA*-relationships.

covering requirement for them, a new *Engineer* may appear only as a new *Employee* and a *Manager* may appear only if the corresponding *Employee* disappears. It means that the *isA*-arrow $isA_E : Engineer \to Employee$ is ins-static and $isA_M : Manager \to Employee$ is del-static in the sense of definition in Section 3.3.8.

4.2.2. Generalization

4.2.2.1. Example. As noted in [1], the construct usually called generalization assumes some essential differences in semantics in comparison with specialization. When we speak about generalization, we usually assume that first we have a few different object classes, say, *Engineer* and *Manager*, for which we then discover some common properties we want to reason about. To do that, we merge these classes into a single class, say, *Employee*, and attribute the common properties in question to it. However, we also want to keep the original classes included into the merge via injective arrows, say, $g_E : Engineer \to Employee$ and $g_M : Manager \to Employee$. Thus, generalization appears as a diagram operation as shown in the fourth row in Fig. 9. In the static set framework, it is nothing but an application of the coproduct operations described in Section 2.3.5 to the nodes *Engineer* and *Manager*. However, in the varset framework we have a more refined picture: because transitions of objects between subclasses are supposed to be prohibited, the subclasses are *globally* disjoint in the sense of definition in Section 3.4.3 and, in addition, a special compatibility with object's deletions-from/insertions-in subclasses holds. Namely, if an *Engineer*- or *Manager*-object is deleted/inserted then the corresponding *Employee*-object is also deleted/inserted. In other words, the inclusion arrows are [`del–ins`]-static arrows as defined in Section 3.3.8(iii).

However, our analysis is still incomplete. A fundamental distinction of generalization from specialization (also noted and stressed in [1]) is that object identity of the objects in the generalization class is derived from object identities in the generalizable (sub)classes. That is, in our example, the *same*-relations for the class *Employee* are derived from the *same*-relations of classes *Engineer* and *Manager*. The way of derivation is evident: we enter two partially defined 1–1 arrows,

$$f_E : Employee \to Engineer \quad \text{and} \quad f_M : Employee \to Manager$$

inverse to "*isA*"-injections g_E, g_M respectively, and for any moments $t \leqslant u$ and any object states $e' \in [\![Employee]\!]^t$, $e'' \in [\![Employee]\!]$" define

$$\begin{aligned}&(e', e'') \in {}^t[\![Employee]\!]^u \text{ iff}\\ &\quad \text{either } e' \in \mathrm{Dom}([\![f_E]\!]^t), e'' \in \mathrm{Dom}([\![f_E]\!]^u) \text{ and } (e'.[\![f_E]\!]^t, e''.[\![f_E]\!]^u) \in {}^t[\![Engineer]\!]^u \qquad (14)\\ &\quad \text{or} \quad e' \in \mathrm{Dom}([\![f_M]\!]^t), e'' \in \mathrm{Dom}([\![f_M]\!]^u) \text{ and } (e'.[\![f_M]\!]^t, e''.[\![f_M]\!]^u) \in {}^t[\![Manager]\!]^u.\end{aligned}$$

Such a way of object identification might be called *disjunctive* or $+$-*identification* in contrast to conjunctive $\times$-identification entered in Section 4.1.1.1, see, e.g., (4). So, by the statement (14) the pair of arrows (f_E, f_M) is declared to be the $+$-*identifier*, or $+$-*key*, to the class *Employee* (note the marker [`ID`$^+$] in the sketch in the fourth row in Fig. 9).

4.2.2.2. Subgeneralization. It is clear that the construct of generalization is in much dual to aggregation but there is an essential difference. While the extent of the *same*-relations for an

aggregation-class is derived, the extent of the very class is a basic item and must be set independently. As for generalization construct, both, the extent of a generalization-class and the extent of its *same*-relations are derived items and, in fact, generalization is an operation precisely dual to the operation of (Cartesian) products over varsets. So, aggregations might be called subproducts of varsets and, thus, the dual to aggregation should be a construct of taking a subset of the entire generalization set considered above. It seems such *subgeneralizations* were not considered in the literature while they may appear in practice quite naturally.

For example, the user might be interested in a subclass *neighborEmployee* of the class *Employee* whose members are those employee who can reach their working places in less than 30 min. The extent of this class depends on employee's addresses, transportation means they use, traffic or weather conditions etc. Since it is unlikely that all these factors are traced and immediately accessible to the user, then the extent of class *neighborEmployee* is to be stored rather than computed. So, all the three nodes of the bottom sketch in Fig. 9 denoted basic varsets. However, object identity of *neighborEmployee*-objects is derived, that is, *same*-relations are computable from *same*-relations for classes *Engineer* and *Manager* as it was specified in (14). Correspondingly, the node *neighborEmployee* in the sketch is double-framed to point out that the class is ontologically derived (weak), and the way of derivation is specified by declaring the pair (f_E, f_M) to be the +-identifier of the class—note the marker $[\mathtt{ID}^+]$ in the sketch.

4.2.2.3. Formal definition. An object class (varset) C is called (*sub*)*generalization* of classes $D_1, \ldots, D_n$, $n \geqslant 2$, if there is a family of single-valued and partial yet jointly total and disjoint variable mappings $(f_i : C \to D_i;\ i = 1, \ldots, n)$, such that C's interstate relations are derived from those of D_i's as follows: for any time moments $t \leqslant u$ and any object states $c' \in [\![C]\!]^t$, $c'' \in [\![C]\!]^u$,

$$(c', c'') \in {}^t[\![C]\!]^u \text{ iff, by definition, } c' \in \mathrm{Dom}([\![f_i]\!]^t) \quad \text{and}$$
$$c'' \in \mathrm{Dom}([\![f_i]\!]^u) \text{ for some } i \leqslant n \quad \text{and } (c'.[\![f_i]\!]^t, c''.[\![f_i]\!]^u) \in {}^t[\![D_i]\!]^u. \tag{15}$$

Note, there can be only one such an index i since the family is disjoint. Then the arrow family $(f_i, i = 1, \ldots, n)$ is called the +-*identifier* of class C, or else the (*primary*)+-key to class C; the way of identification is called *disjunctive*.

It follows immediately from the definition that for any time moment t, all mappings $[\![f_i]\!]^t$ are injections. To see this it is sufficient to put $u = t$ and apply reflexivity of *same*-relations.

4.2.2.4. Remark. Arrows constituting the key to a generalization class are inverse to inclusions of the generalizable subclasses into it. This is a principle aspect of the issue and it was noted and explicated in [1].

Actually, the richness of types of *isA*-arrows is not exhausted by specialization and generalization described above. Several other possible situations are shown in Fig. 9 and the reader can evaluate the flexibility of the sketch language.

4.3. Conceptual modeling via sketches, II—dynamic setting

We begin with application of the dynamic constructs we have entered above to the example considered in Section 2.4 (Fig. 4).

4.3.1. Example continued and completed

We return to our sample universe presented in Fig. 4. A sketch S_{dyn} modeling the universe in the framework of *variable* sets and mappings is presented in Fig. 10. It is instructive to compare it, on one hand, with the ER- and UML-diagrams in Fig. 4(a) and (b) presenting the universe in an informal CM-slang, and on the other hand, with the sketch S_{sta} in Fig. 4(c) specifying the universe formally but in the static framework of ordinary sets and mappings.

(1) The most fundamental distinction between S_{sta} and S_{dyn} is that identifiers (keys) to object classes are explicitly designated in the dynamic sketch, they are denoted by marker $[ID^{\times}]$ and $[ID^{+}]$. Correspondingly, object classes whose identification depends on other classes become ontologically derived (weak), they are denoted by double-framed nodes. Particularly, ontologically derived are classes *Association* and *Child* by, respectively, disjunctive $+$-identification and conjunctive $\times$-identification; in this way the two weak-entity declarations in the ER-diagram are formally explicated and made precise. Similarly, $\times$-identification declared for the node *Family* makes *Family*-objects aggregations of *Person*-objects additionally qualified with *mDate*-values. Node *Membership* is also declared ontologically derived by $\times$-identification; a peculiarity of the dynamic sketch here is that *Membership*-objects are additionally declared as legacyless (Section 4.1.2).

So, in the ER-diagram we have three relationship nodes—*Membership*, *Married*, *Parentship*—which are refined on the UML-diagram as association-class, aggregation and pure association link; however, the UML does not provide formal semantics for these constructs. The static sketch S_{sta} does have a formal semantics but does not capture the distinction between association class and aggregation. Only the dynamic sketch S_{dyn} seems to provide a really adequate and formal description of the issue.

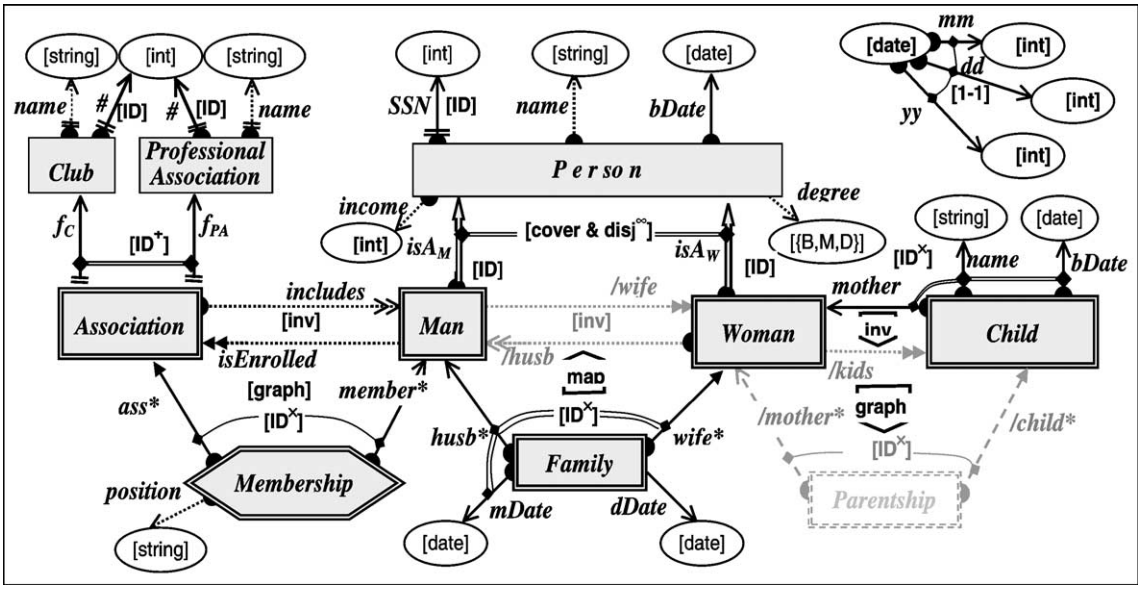


Fig. 10. Sketch model of the universe, completed. Dynamic keyed sketch, $S_{dyn} = \mathbf{ske}[M^{ER}_{dyn}(D') + M^{UML}_{dyn}(D'')]$ (compare with static sketch S_{sta} in Fig. 4).

(2) The dynamic sketch also captures the difference between the fragment (*Person*, *Man*, *Woman*) seen as specialization and a similar fragment (*Association*, *Club*, *profAssociation*) seen as generalization. This difference is presented in the ER-diagram by using different *isA*-arrows (incorporated from the IFO-metamodel) while the UML does not distinguish between them.

(3) One more peculiarity of the dynamic sketch in comparison with its static counterpart is in a different interpretation of the oval shape of value-domain nodes. In the static sketch it was just an additional visualization sugar to decorate nodes marked with special predicate labels like [string], [integer] and the like. In the dynamic sketch, the oval shape is a predicate label declaring the node (varset) to be consisting of static objects in the sense of Section 3.2.1.

(4) Finally, three more distinctions from the static sketch are present in the dynamic sketch $\mathsf{S}_{\mathrm{dyn}}$:

- the subclasses *Man* and *Woman* are more than disjoint, they are globally disjoint,
- the attributes *SSN*, *name* (of *Club*) and *name* (of *profAssociation*) are more than keys, they are global keys,
- some arrows are declared static (frozen) and their bodies are solid lines while others denote changeable mappings and their bodies are dotted.

Particularly, since *name* attributes of classes *Club* and *profAssociation* are not static, to properly identify objects of these classes, we introduce their globally unique static numbers (surrogates) #'s. Then *Association*-objects are identified by pairs $(*,\#)$ with $*$ a symbol either *C* or *PA* and # a number.

4.3.2. Abstract syntax for conceptual modeling

Examples we have considered manifest an abstract syntax framework for CM where any conceptual data schema (model) is just a (visualization of a) keyed sketch. In more detail, this means the following.

(1) First of all, a conceptual model, or schema, S is a directed graph $G(\mathsf{S})$ whose nodes denote variable sets and arrows denote variable mappings. In addition, some diagrams in $G(\mathsf{S})$ are marked with predicate labels taken from a predefined signature Σ, for example, Σ may contain labels [1–1], $[1\text{–}1]^{\infty}$, [cov], [disj], $[\mathtt{disj}]^{\infty}\ldots$ we described above. Importantly, a signature should necessary contain a set of special node labels (like $[\mathtt{string}], [\mathtt{integer}], \ldots$) which declare nodes they mark as *value domains* with predefined semantics; nodes not marked with labels from this set are considered to be *object classes*. Finally, for each class node a family of outgoing arrows is declared to be either the $\times$-key or the $+$-key to the node. These key arrows together form a special subgraph $G_{\mathrm{ID}}(\mathsf{S})$ of graph $G(\mathsf{S})$.

(2) The key subgraph constitutes the way objects in the universe are identified. For example, following along the [ID]-arrows going out of node *Family* in Fig. 10, we see that Family-objects are identified by triples of values $(mDate, husb^{*} \Rightarrow isA_M \Rightarrow SSN, wife^{*} \Rightarrow isA_W \Rightarrow SSN)$ where $\Rightarrow$ denotes arrow composition (*isA*-arrows are, in fact, dummy). Similarly, *Membership*-objects are identified by pairs of values $(ass^{*} \Rightarrow [f_C \Rightarrow \# | f_{PA} \Rightarrow \#], member^{*} \Rightarrow isA_M \Rightarrow SSN)$. In order to have *finitary* identification of objects by values we must require acyclicity of the key graph $G_{\mathrm{ID}}(\mathsf{S})$.

A precise definition of what is a *well-formed* schema can be formulated: the main requirement are acyclicity of the key graph and terminality of the set of value-domain nodes w.r.t. the key graph [12]. Then, an important result about well-formed schemas can be proven: in any extension

of a well-keyed schema S, objects are uniquely identified by values in a *finitary* way, that is, every object is uniquely identified by a finite tuple of values.

(3) Given a conceptual data schema (keyed sketch) S, its specialization, generalization and aggregation networks are subgraphs of the key subgraph, $G_{\mathrm{Spec}}(\mathsf{S}) \subset G_{\mathrm{ID}}(\mathsf{S})$, $G_{\mathrm{Gen}}(\mathsf{S}) \subset G_{\mathrm{ID}}(\mathsf{S})$ and $G_{\mathrm{Aggr}}(\mathsf{S}) \subset G_{\mathrm{ID}}(\mathsf{S})$. Then a well-formed schema has no cycles in either of these networks, in particular, in the *isA*- and aggregation networks—the two well-known conditions in CM and OOAD (see, e.g., [37]). Note that while usually these conditions appear as independent basic postulates, in our framework they are both derived from the well-understood requirement of finitary identification of objects. And of course, absence of cycles in $G_{\mathrm{ID}}(\mathsf{S})$ is a much more general condition than acyclicity of $G_{\mathrm{isA}}(\mathsf{S})$ or $G_{\mathrm{Aggr}}(\mathsf{S})$ separately.

Note, in practice we often deal with conceptual schemas containing multiple *isA*-sources, that is, nodes with a few *isA*-arrows going out of them. It may seem that in such a case the uniqueness of object identification is violated. However, this is not the case: in the accurate formulation, a class is not allowed to have two different *deep* primary keys whereas several *shallow* primary keys are admissible if the corresponding paths commutes, that is, their semantic interpretations are equal mappings. One can verify that if a semantic schema contains an *isA*-multisource, then the condition of commutativity is always assumed by default.

Some more details about well-formedness of keyed sketches can be found in [12].

5. Mathematics of conceptual modeling—discussion

The present section is mainly methodological and a bit speculative. Here we will try to place the framework we described above in a wider mathematical context and outline a mathematical view on the CM-problems. On the other hand, we will try to relate our framework to some general trends in CM in a historical retrospective. In fact, both, category theory in 60s and conceptual modeling (in the context of database design) in 70-80s, have been approaching basically the same problem area of studying/specifying universes of sets but of course with different goals and in different contexts. Such cases of meeting different intellectual attitudes are always interesting, their study is instructive and interaction might be very fruitful.

5.1. What logic is relevant for conceptual modeling

5.1.1. The two basic specification frameworks

Let D be a diagram—a conceptual model of some universe $\mathscr{U}$ expressed in some diagrammatic notation. Let $\mathscr{M}(\mathsf{D})$ denotes the intended semantic meaning of D: it is some description of $\mathscr{U}$ in terms of objects, values, associations, aggregations, *isA*-relationships etc. Though rather abstract, such a description is informal. Formal semantics for D is a refined reformulation of $\mathscr{M}(\mathsf{D})$ in entirely abstract terms (like set, mapping, relation) of some formal language. Developing formal languages is a subject of mathematics and to date two main general frameworks are known.

One of them is adopted in familiar formalizations of first order logic and widely spread. This framework is based on the notion of set of abstract uninterpreted elements for which certain predicates and operations are defined. Then complex (derived) operations and predicates are expressed in a linear notation with logical connectives and quantifiers. Thus, formal specifications

are sets of declarations about strings of elements and we call the framework *element-wise string logic*, ElmSL. Formal refinement of $\mathscr{M}(\mathsf{D})$ in this approach, $M_{\mathrm{ElmSL}}(\mathsf{D})$, is specified by a set T of logical formulas (a *theory*, in terms of mathematical logic):

$$\mathsf{T}(\mathsf{D}) = \mathbf{thr}M_{\mathrm{ElmSL}}(\mathsf{D}).$$

The second framework was developed in CT and is based on the notion of universe of abstract sets and mappings between them. In contrast to ElmSL, predicate and operations considered here are predicate and operations over configurations (diagrams) of sets and mappings while elements as such are hidden. Thus, in this framework, formal specifications are graphs denoting sets and mappings, in which certain diagrams are marked with predicate labels.We call the entire framework *arrow diagram logic*, ArrDL. Formal refinement of $\mathscr{M}(\mathsf{D})$ in this approach, $M_{\mathrm{ArrDL}}(\mathsf{D})$, is specified by a sketch (graphical theory) S:

$$\mathsf{S}(\mathsf{D}) = \mathbf{ske}M_{\mathrm{ArrDL}}(\mathsf{D}).$$

It was proven that both frameworks are equally, and absolutely, expressive: any formal construction can be specified by either a theory in ElmSL or by a sketch in ArrDL. Thus, formal semantic meaning of any CM-diagram D can be specified by either a theory $\mathsf{T}(\mathsf{D})$ or a sketch $\mathsf{S}(\mathsf{D})$ as well. However, there is a big difference between a principle possibility to specify something and how apt the resulting specification will be for work and use. For example, any program can be specified in terms of Turing machine yet we have observed a dramatic progress in software towards equally expressive yet more manageable and natural programming languages. In this sense, ElmSL and ArrDL are very different specification frameworks. Our main thesis here is that ArrDL is perfectly suitable for approaching the CM-problems while ElmSL is not.

5.1.2. ArrDL vs. ElmSL

Two major problems appear with specifying the meaning $\mathscr{M}(\mathsf{D})$ in ElmSL. First, while D normally presupposes graph-based viewing the world, ElmSL-specifications consist of strings and thus there is an essential *syntactic gap* between the concrete syntax, diagram D, and the abstract specification, $\mathsf{T}(\mathsf{D})$, underlying it.

The second problem is even more essential. It is generated by a *semantic gap* between the meaning of a typical CM-diagram D and regular semantics of ElmSL-theories. The point is that ElmSL presupposes viewing the world in terms of elements while normally diagram D describes the world in terms of sets (object classes, consider D as an ER-diagram or UML-diagram). This major semantic discrepancy leads to a few particular problems with using ElmSL in conceptual modeling.

For example, it is an ordinary thing in CM when a set of relationships (associations) is of interest by itself and then it is reasonable to consider it as an association class: a regular object class bearing, in addition, an association structure. However, this natural setting is foreign for ElmSL where relations are not basic domains (sorts) with extra structure but something built over domains and conceptually different from them. [18] In other words, it is not customary for ElmSL

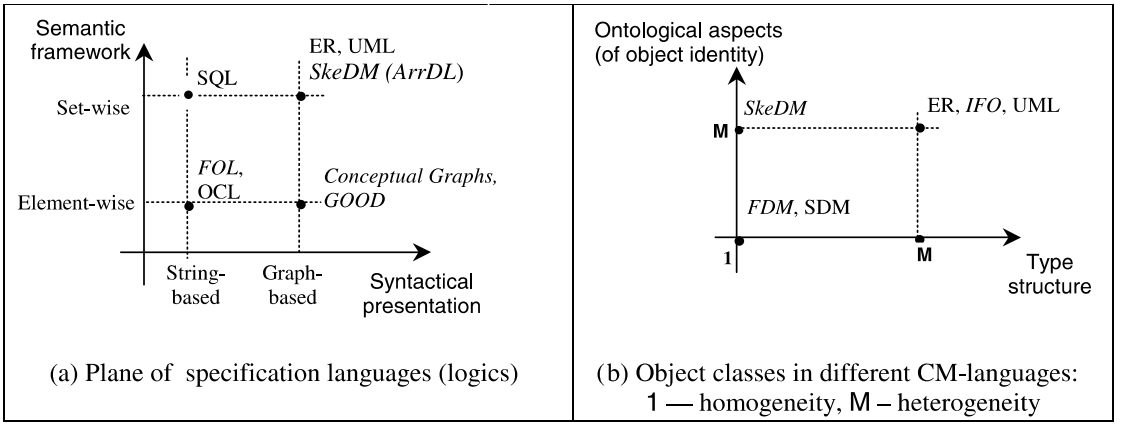
[18] In a bit more detail, a regular ElmSL-structure is a collection of basic sets (sorts, domains) over which a family of relations and operations is defined. There is a fundamental difference between a sort set consisting of uninterpreted elements and a relation consisting of tuples.

to consider elements of relations as just elements forgetting their tuple structure; it may be said that this structure is not encapsulated. The same problem occurs with other complex types: they are not encapsulated in ElmSL.

For another instance of semantic discrepancy between CM-diagrams and ElmSL-specifications, consider the well known phenomenon of semantic relativism. For example, what one user considers to be an ordinary object class, another user views as an association class. If both views are specified in ElmSL, then normally the class will be modelled by a basic sort set in the first view and by a relation in the second view. To relate the two views, we need to map (interpret) a basic sort of the first view into a relationship set in the second but such interpretations of theories can present a serious problem for conventional ElmSL's, for example, they are not allowed in FOL.

Finally, the basic feature of the elements of real world sets (classes)—their object identity—is hard to model within ElmSL where object identity is degenerated into distinction of individual elements within sets. In general, a consistent attempt to incorporate objects into usual elementwise relational languages leads to intricate formalisms like, e.g., O-logic and F-logic [32].

In a sharp contrast with ElmSL, ArrDL is perfectly suited for syntactically graph-based and semantically set-wise specifications (see Fig. 11(a)). In ArrDL, a structure σ of elements of set X is set by declaring a certain predicate P_σ for a certain diagram D_σ of sets and mappings adjoint to X. For example, as we have seen in Section 2.3.1, instead of saying that elements of set X are tuples, in ArrDL one enters the predicate declaration $P_{[1-1]}[D_{\text{tuple}}(X)]$ for the corresponding arrow diagram $D_{\text{tuple}}(X)$ adjoint to X. Such a diagram can be considered as an arrow interface to the class X and so the idea of the approach can be phrased as that objects show themselves only through arrow diagram interfaces. It goes in evident parallel with a basic OO principle of encapsulation, the more so that arrows can be treated as formal counterparts of references and attributes (as for the methods, they are also treated by arrows but in a quite another space). So, in ArrDL all sets are considered plain and homogeneous, and it allows to manage the issue of semantic relativism. All semantic conflicts are reduced to the only type of relativity when data considered basic in one



(a) Plane of specification languages (logics)

(b) Object classes in different CM-languages: 1 — homogeneity, M – heterogeneity

Fig. 11. Sketch data model (SkeDM) in general mathematical (a) and conceptual modeling (b) landscapes. Abbreviations used: FOL—first order logic, FDM—functional data model [38], OCL—object constraint language [26], SDM—semantic data model [24], GOOD—graph-oriented object database model [23], IFO—data model developed in [1], where italic indicates that a formal semantics is available.

view are considered derived in another view (see [6]) and, of course, when we consider interpretations between views, mapping basic items (nodes and arrows) into derived ones is not prohibited.

5.2. Conceptual modeling and category theory

5.2.1. Conceptual models as quasi-toposes

In the view we have developed, building a conceptual model of a universe amounts to presenting the universe as a collection of variable sets and mappings subjected to certain diagram predicates and carrying certain diagram operations.

Briefly, a conceptual model of a universe is its presentation as a *varset universe*. Such universes were studied in CT in a very general and abstract setting. In fact, a well-elaborated hierarchy was built where universes are differentiated in function of what operations over their objects are legitimate and which properties the operations have.

In the top part of the hierarchy are universes where all the (abstract counterparts of) ordinary operations over sets can be performed: simple, like intersection and union, and more complex like powerset or lambda abstraction. Such universes were called *toposes* and actually they range from proper toposes, where the complement to a subset may not exist, to classical Boolean toposes like the topos **Sets** of ordinary sets and functions between them. [19] Below in the hierarchy are universes where only some of standard set-theoretical constructs are legitimate, or/and standard constructions have some unusual properties: roughly, the less properties of **Sets** are guaranteed, the lower the universe in the hierarchy. For the present discussion, we will call any universe of set-like objects a *quasi-topos* or *CM-topos* while toposes in the precise sense defined in CT will be called *CT-toposes*.

In the database context, operations on set-like objects are nothing but queries. Then, the hierarchy mentioned above is a hierarchy of query languages (and associated hierarchy of constraints languages with constraints being equalities between particular queries).

5.2.2. Powerset operation in CM-toposes

Topos studies in CT were motivated mainly by pure mathematical problems. In particular, an essential component of a CT-topos is the powerset operation: for any object (to be thought of as a varset) X there is another object $\mathbf{Pow}X$ having special properties which allow to think it of as the varset of subsets of X. This is extremely non-effective operation and, of course, set universes considered in CM have a very restricted version of powerset: for each binary relation $R \subset X \times Y$ there is a set $\mathbf{Pow}_R Y$ and a mapping $\mathbf{f}_R : X \to \mathbf{Pow}_R Y$ with properties which are abstract counterparts of the ordinary interpretation when $x.\mathbf{f}_R \stackrel{\text{def}}{=} \{y \in Y | R(x,y)\}$. In CT-toposes, all such $\mathbf{Pow}_R Y$'s are subsets of the same $\mathbf{Pow}Y$ while in CM we deal with quasi-toposes where such $\mathbf{Pow}Y$ may not exist.

[19] In a bit more detail, topos is an abstract mathematical structure like group or vector space. A particular universe may then be an instance of the topos structure, that is, *a topos*, if it possesses all the necessary operations the definition of topos requires.

5.2.3. Varsets in CM-toposes

Another essential distinction between CM- and CT-toposes is in the very notion of variable set. Variable sets considered in CT are supposed to have very special *same*-relations: they are totally defined single-valued mappings (such varsets are called *presheaves*, see [2] for definition and discussion). However, as we have seen, CM does need the generalization of presheaves to varsets with arbitrary *same*-relations. [20]

5.2.4. Object classes and value-domains in CM-toposes

An important peculiarity of universes to be dealt with in CM is that they consist of the two fundamentally different kinds of varsets: value domains and object classes. This property of CM-universes is specially stressed and formally captured in such mathematical frameworks as *meta-finite model theory* [21] and *hidden algebra* [19]. It may be also easily explicated in the sketch framework.

Any signature of diagram predicates and operations is assumed to contain a special set of node labels like $[\mathtt{string}], [\mathtt{integer}], \ldots$; more accurately, they are labels of 0-ary diagram operations producing a node—the value domain of the operation. The graph $G(\mathsf{S})$ of sketch S specifying a universe contains a subgraph $G_{\mathrm{Dom}}(\mathsf{S}) \subset G(\mathsf{S})$ of (nodes marked as) value domains and mappings (arrows) between them, the latter are operations on values supported by the computer. A natural semantics for this part of the sketch is the so called *initial* algebraic semantics (first proposed in [20] and then) well studied in abstract data types theories (see, e.g., [43], the reader may also take a look at [22] for a sketch treatment of initial semantics).

As for object classes, their extension is not bound to a special semantic interpretation (so called *loose* semantics) while their behavior is modeled through state spaces with methods and attributes (coalgebraic operations) defined on them. A special semantics for this later component is the so called *final* semantics dual to initial semantics (see, e.g., [27]).

The value domain subgraph $G_{\mathrm{Dom}}(\mathsf{S})$ plays a special terminal role w.r.t. object class nodes: in a well-formed sketch, the compositive closure of the key subgraph $G_{\mathrm{ID}}(\mathsf{S})$ provides finite value-based identification of objects of any class node [12].

Thus, to summarize, in CM we deal with a special version of a general mathematical structure of topos. Its peculiarities are (i) a restricted powerset operation, (ii) varsets with arbitrary *same*-relations, (iii) two major kinds of varsets—object classes and value domains.

5.2.5. Remark on toposes

There is a vast literature devoted to topos theory, see, e.g., [2] for commented references. An exciting fact of topos theory is that many constructions appearing in the two seemingly very different mathematical areas—algebraic geometry and logic—turn out just different particular instances of the same abstract structure. It follows from our considerations above that there is one more substantial interpretation of the topos formalism—the CM interpretation in terms of object identity. It allows to apply (constructive versions of) methods and techniques developed in CT

[20] That leads to an essentially different logic of reasoning about varsets [18].

to CM-problems thus providing a powerful mathematical framework for the latter. A notable example can be found in [36] where the problem of semantic equivalence between conceptual models is treated in a mathematically rigorous way.

An important aspect of the topos view on conceptual models is that topos is an essentially algebraic structure. Then the task of building a conceptual model of a universe amounts to a well known algebraic problem—building a finite presentation of infinite algebraic structure. This opens the door to applications of powerful algebraic methods/techniques elaborated in CT, particularly, *diagram chasing*. However, an essentially different context needs essential adjustments and development of the framework. Some details on what and why should be done to adjust machinery developed in CT to CM-problems, and elaboration of the first steps on the way (together with references to related CT-literature), can be found in [10].

5.3. Relation to other works

The idea to use graph-based languages for data definition and manipulation, and support them with precise mathematical machineries, is evident and attractive. Not surprising that there were a few attempts (of different degrees of completeness) to realize it and, in particular, to apply CT to these problems. Without any pretension to completeness, we mention some of them which are either most close to our approach or underline its peculiarities by comparison.

5.3.1. Conceptual modeling via sketches

The most close to ours is the approach developed by M. Johnson, C.N.G. Dampney and their coauthors, see [30] for a precise mathematical description and references, and [29] for a more CM-oriented presentation. They also consider ER-diagrams as sketches of a special kind (ER-sketches), and queries as diagram operations over sketches, but there are some essential differences. An important one is that they work with some fixed set (signature) of diagram predicates and operations (setting a so called *lextensive category*), that is, actually fix a query and constraint language, while for us signature is a user-defined parameter. In addition, their signature does not contain any kind of powerset operation while we insist that a special constructive version of powerset operation (discussed in Section 5.2.2) is legitimate and presents an important for CM construct of grouping.

A crucial distinction is that they work in the static set framework for semantics and thus have no means to explicate adequately such semantic constructs as object identity, aggregation, composition, *isA*-relationship and the like. Also, in their framework the distinction between object classes and value domains is of technical rather than principle nature. To summarize, keyed sketches described in the present paper and keyed sketches of M. Johnson et al. are different syntactically and have fundamentally different semantics.

A different approach was presented in [36] where a particular kind of sketches is also considered as a conceptual data modeling paradigm. The focus of that paper is in semantic equivalence of conceptual data schemas and indeed, the kind of sketches they use allows them to prove non-trivial results about semantic equivalence. However, as a conceptual modeling language, sketches used in [36] are rather restrictive.

5.3.2. Another attempt to apply category theory to CM

There are also a few works of A. ter Hoftsede, E. Lippe and their coauthors where a direct application of CT to CM-problems is declared, see [40] for a detailed presentation and references. [21] In contrast to works mentioned above which are strict and, in fact, mathematical, the framework of ter Hofstede et al. is in much declarative and informal. In syntax, what they call a type graph is a deficient sketch where some of basic diagram predicates (say, of tuple and grouping type) are absent and authors try to emulate them by special arrow properties and some additional structure defined over the carrying graph. In fact, they miss a general notion of diagram predicate and try to emulate it in a somewhat artificial way. A special emphasis is made on *configurable* semantics that is indeed a great advantage of the categorical framework (we discuss it in Appendix A.4). But again, among different possible semantic universes they consider, the varset semantics is absent and thus an adequate treatment of advanced semantic constructs turns out beyond the framework.

5.3.3. GOOD, G-Log, GraphLog

These are well-known graphical data definition and manipulation languages. They are based on element-wise specifications invented in [23] and further developed in a number of works (see [35] for references); among them there is a categorical treatment [42]. These works essentially differ form the present paper in that they (i) address mainly the issue of query language completeness and (ii) are based on fundamentally different semantic foundations of element-wise considerations. From the view point of the present paper, such an approach is too much of implementation while sketches and diagram algebra over sketches are declarative.

5.4. Conceptual modeling in the light of sketches—a historical retrospective

Certainly, the idea of considering a universe of discourse as a set universe, in particular, object classes as plain homogeneous sets, is not new in CM. Moreover, it was the principal one for well-known Functional Data Model (FDM) [38], Semantic Database Model (SDM) [24] and other semantic models *stressing the use of attributes to interrelate objects*. [22] However, as we have seen in Section 2, classes-as-sets formulation is only a half of the story, the second—crucial—component is to express internal type structures of classes externally via arrow diagram predicates. It seems this idea was never recognized in CM in a explicit and systematic form though, of course, a few ad hoc fragmentary uses of diagram predicates can be found in different CM-languages including the UML.

We have seen that ontological status of objects in a class is determined by its defining identifier with certain structural and temporal properties. Thus, object ontology is placed in a structural–temporal space and different CM-frameworks could be roughly classified in function of whether all object classes are a priori assumed or not assumed having the same type structure and the same nature of object identity. In some conventional two-dimensional plane (Fig. 11(b)), it gives rise to

[21] An early attempt to apply sketches directly to CM in a similar way can be found in [34].

[22] This characteristics refers to a classification in [25] where all semantic models were divided into *attribute-oriented* and *type-constructor-oriented* in function of the preferred way of interrelating objects. In the latter kind of models (e.g., IFO [1]), sets of different types (tuple, grouping) are considered.

the four classification points—general possible orientations of CM-frameworks. Several well-known semantic data models, and the sketch data model (SkeDM) as well, can be attributed to these orientations as shown on the figure.

Of course, Fig. 11(b) presents a very rough view and some reservations should be made. One is that due to the machinery of diagram predicates, types in SkeDM become implicit rather than disappear at all. Another one is that SkeDM allows to model a variety of object ontologies where entities and relationships are only the two extreme points (see [9] for details). So, a rough notion of relations between sketches and other semantic models can be expressed by a sort of formula-slogan

$$\text{SkeDM} = (\text{FDM} + \text{DiagramPredicates}) \times \text{ER}.$$

Without the temporal dimension, the plane in Fig. 11(b) is degenerated into the structural scale with two points—these are nothing but the two philosophies of conceptual modeling distinguished in the survey [25]: attribute-oriented and type-constructor-oriented (we have already mentioned them). Correspondingly, the difference between entities and relationships is degenerated into only the structural one, and so the real sense of the ER-model disappears. Contrariwise, we believe that the true value of Chen's ER-proposal was in designating the ontological (rather than structural) difference between entities and relationships. It seems that this point is not recognized in the literature (at any rate, in a explicit form).

In a sense, the sketch approach to CM can be considered as a logically consistent development of the ER-idea up to a complete form: sketches as such is a completion of the structural component—the ER-diagram notation, and the variable sets semantics for sketches is a completion of the temporal ontological component—the entity-relationship dichotomy. For example, the extensive in-width development of the ER-components in the UML, where a whole variety of associations between classes is introduced, fits in well with the sketch framework [13]. We believe that wide use of ER- and now UML-diagrams in the practice of conceptual modeling has already created certain prerequisites of sketch-based CM. On the other hand, the sketch treatment sets a proper mathematical foundation under CM and so makes it a really well-formed engineering discipline. In this sense, category theory becomes a calculus for CM (very much like the ordinary calculus is for mechanical and electrical engineering), and justification of this statement is maybe the most important general result of the paper.

6. Conclusion

The results of the paper are twofold: semantic and syntactical. On the semantic side, we have proposed a formal semantics for object identity based on the notion of variable set. In this framework we then built a formal semantics for major constructs of conceptual modeling, such as association, aggregation, generalization, *isA*- and *isPartOf*-relationships. We do not assert that our formal constructions give a fully adequate semantics for the constructs in question: objects' behavior, which can be an essential component of semantics, is a priori beyond our formal framework. So, what we really did is elaboration of how much of intuition underlying the constructs above is associated with object identification, and hence can be formally explicated in the variable set framework. The paper shows that this identification component is indeed essential,

and so the variable set treatment of object classes is an important part of the semantic picture. To complete the latter, objects' state spaces have to be added to interpretation of classes, providing one more—behavioral—dimension in semantics.

In a bit more detail, we mean, roughly, the following. For a given logical time moment (global state of the system), t, semantics of an object class C is given by a mapping

$$state^t_C : [\![C]\!]^t \to \langle\!\langle C \rangle\!\rangle,$$

where $[\![C]\!]^t$ is extension of C at the moment t, $\langle\!\langle C \rangle\!\rangle$ is the *state space* of C, that is, some set of internal states of C-objects, and $state^t_C$ maps an object $o \in [\![C]\!]^t$ into its state at moment $t, o.state^t_C \in \langle\!\langle C \rangle\!\rangle$. Note, the state space does not depend on t and changeability of objects is modeled by changeability of the mapping $state^t_C$. In addition, as we have seen in the paper, extensions of classes are also changeable, and if two moments t, u are logically related (by a possible transition), $t \leqslant u$, then sets $[\![C]\!]^t$ and $[\![C]\!]^u$ are interrelated by a binary relation ${}^t[\![C]\!]^u$ of object identity.

Thus, semantic relationships between classes C_i, $i = 1, \ldots, n$ have the two sides. One, associated with object identity, involves only variable sets $[\![C_i]\!]^t$s and mappings between them while state spaces $\langle\!\langle C_i \rangle\!\rangle$s are disregarded—just this projection of the entire picture was considered in the paper. The other side of semantic relationships is associated with objects' behavior and involves state spaces $\langle\!\langle C_i \rangle\!\rangle$s together with attributes and methods defined on them. This component of the framework can be really essential for modeling semantic relationships between classes. For example, declaring a class C to be a composition of classes C_i means, among other things, that its state space is composed from state spaces of the components via Cartesian product: $\langle\!\langle C \rangle\!\rangle = \langle\!\langle C_1 \rangle\!\rangle \times \cdots \times \langle\!\langle C_n \rangle\!\rangle$ and thus class C has projection methods $p_i : \langle\!\langle C \rangle\!\rangle \to \langle\!\langle C_i \rangle\!\rangle$ $i = 1, \ldots, n$. Moreover, (at least some of) C's attributes and methods must be somehow derived from C_i's attributes and methods via projections mappings connecting the state spaces. In this way C's properties are (at least in part) determined by the components properties, which is considered to be an essential aspect of the composition construct (see, e.g., [33]).

On the syntactical side, all our semantic constructions are specified in a format which we have called *sketch*, borrowing this term from category theory. A sketch is a directed graph in which some diagrams are marked by predicate labels taken from a predefined signature. Surprisingly, but this extremely compact specification pattern is extremely expressive so that any formal construction can be specified by a sketch. In this sense the sketch format can well serve as a universal abstract syntax for the entire CM field, and we have shown in the paper how it works for specifying main CM-constructs. It appears that major notations in CM including ER and UML miss the fundamental notions of diagram predicate and diagram operation, at least, underestimate their fundamental role for making diagrammatic languages consistent and mathematically correct.

In particular, with the use of these notions, some issues of conceptual modeling can be essentially clarified even on the much poorer level of the static set semantics when we deal with extensions $[\![C]\!]$s irrespectively to their change in time. For example, we have shown that presentation of relations by families of jointly monic mappings is a proper approach to modeling structural aspects of associations. We have also demonstrated that constraints involving diagram operations can be essential in conceptual models. More details on how the static set semantics is treated in the diagram logic and algebra framework can be found in [15].

A great advantage of the sketch language for CM is that different semantics frameworks (static sets, variable sets, variable sets endowed with state spaces) are based on the same syntax. Going from poorer to richer semantic frameworks results in richer signatures of diagram predicates and operations but the very syntactical format remains the same: nodes, arrows, marked diagrams. So, syntactical patterns we have proposed in the paper for specifying CM-constructs are also suitable for complete semantics interpretation including object state spaces. Thus, the main thesis of the present paper can be formulated as that *the logic of visual conceptual modeling is the arrow diagram logic*.

Acknowledgements

We are obliged to Ilya Beylin for stimulating criticism and overall discussions of the subject, including notation, terminology and presentation of the material. Special thanks go to George Sheinkman for helpful discussion of object identification in real databases and valuable general remarks and encouragement. Other special thanks go to Vitaly Dzhitenov who transformed the sketch ideology into the programming code; as it happens, a smart implementation helps to understand specification.

We are grateful to Haim Kilov who read the text and made a lot of particular remarks and general comments, which really helped to shape and present the material more correctly. Also, conceptual/business modeling on a whole and some of its particular problems, e.g., semantics of composition, is a recurrent theme of our communication and traces of these discussions can be found in many places of the paper.

We are also grateful to the two anonymous referees who read the first (huge!) draft of the paper and provided useful comments on presentation of the material. Finally, we are obliged to Peter Chen for inviting us to write a journal version of our ER'95 paper, and then for his patience as the journal's Editor-in-Chief and the encouragement we have obtained during endless reengineering of the paper.

Appendix A. Some details on sketch syntax and semantics

We will present some details—not enough to explicate a completely formalizable picture yet enough to give a notion of the main peculiarities of the sketch language, its syntax and semantics. We will do this job for a particular example—sketch $\mathsf{S}'_{\mathrm{sta}}$ in Fig. 2(b) considered in Section 2.1—but it can be well considered generic: other cases can be formalized along the same lines. To simplify notation, this sketch will be denoted by S.

A.1. Syntax—marked diagrams

Any sketch is built in a predefined signature of diagram predicate labels which we will also call *predicate markers*. Each such a marker has its *arity shape* that shows a configuration of nodes and arrows for which the predicate can be declared (that is, the label can be hung). For example, the arity shape of the *inverse arrows* predicate (marker [`inv`], see the top row in Fig. 2(c)) is a graph of two nodes and two arrows between them going into the opposite directions.

The very predicate declaration (in a sketch) is just an assignment of this sketch's nodes and arrows to shape's nodes and arrows (so that their incidence is preserved). For example, in the sketch S in Fig. 2(b′), two such declarations are present:

$$[\mathtt{inv}]\{f \mapsto owns, g \mapsto isOwned\} \quad \text{and} \quad [\mathtt{inv}]\{f \mapsto drives, g \mapsto isDriven\} \tag{16}$$

where f, g denote arrows in the shape (we omit the evident node assignments from the description). In category theory, assignments like above are called *diagrams* and we say that the sketch S contains two *diagrams marked by* the label [inv]. Double-body shape of inclusion arrows and special tails and heads of other arrows are also predicate markers whose arity shapes are trivial one-arrow graphs.

Similarly, with the predicate label [⇑] in the sketch S the following declaration is associated:

$$[\Uparrow]\{f \mapsto isA_M, g \mapsto owns, p \mapsto drives, q \mapsto isA_C\}, \tag{17}$$

see the second row in Fig. 2(c) for the shape of the marker. The same predicate holds for the diagram $[\Uparrow]\{f \mapsto isA_C, g \mapsto isOwned, p \mapsto isDriven, q \mapsto isA_M\}$, but this can be derived from (16) and (17) and thus we omit it in the sketch.

A.2. Visual vs. analytical sketches

How will the user of the sketch S know that the marker [⇑] is hung just on the diagram pointed in (17) and not on another diagram, say

$$\{f \mapsto isA_C, g \mapsto isOwned, p \mapsto isDriven, q \mapsto isA_M\}?$$

(well, as noted above such a declaration would be also true but our concern here is about syntax). The visual image in Fig. 2(b′) is indeed ambiguous in this sense. However, the point is that this image is just a visual presentation of a precise sketch specification hidden in it. When one builds sketches with computer, marked diagrams are naturally set as follows. One first draws the carrying graph of the sketch, then takes a predicate he wants to declare from a predefined signature and, finally, sets a diagram by consequtive clicking an item on the arity shape and then immediately an item in the sketch until all items in the shape are passed (computer will, of course, check compatibility with nodes-arrows incidence). All these data are stored in the computer and actually form a part of the sketch. After it is built, the diagram of a predicate label can be easily revealed by highlighting its items when clicking the label.

So, the visual image in Fig. 2(b′), *visual sketch*, is a visual presentation of precise analytical specification, *analytical sketch*, consisting of (i) a graph (in its turn consisting of nodes and arrows) and (ii) a set of marked diagrams. This specification is presented in Fig. 12 in some ad hoc syntax.

A.3. Sketch semantics

We first fix some universe **Set** of sets and (partially defined multi-valued) mappings with a standard collection of ordinary set-theoretic diagram predicates over it. In particular, the predicate $[\mathtt{inv}]^{\mathbf{Set}}$ may be declared for a pair of mappings going into opposite directions between the same two sets and means that they are mutually inverse (for semantics of diagram predicates in question see the last column in Fig. 2(c)). The predicate $[\Uparrow]^{\mathbf{Set}}$ may be declared for a diagram

```
beginSketch(S):
    beginGraphOf(S):    nodes: Person, Vehicle, Car, Man;
                        arrows: isA_M, owns, isOwned, ...
                          with source(isA_M)=Man, target(isA_M)=Person,
                               source(owns)=Person, target(owns)=Vehicle,
                               ...;
    endGraph
    beginDiagramsOf(S):
                        [inverse]{f ↦ owns, g ↦ isOwned},
                        [inverse]{f ↦ drives, g ↦ isDriven},
                        [semiCommutativity]{f ↦ isA_M, g ↦ owns, p ↦ drives, q ↦ isA_C},
                        [inclusion]{f ↦ isA_M},
                        [inclusion]{f ↦ isA_C},
                        ...;
    endDiagrams
endSketch
```

Fig. 12. Analytical specification of sketch in Fig. 2(b).

consisting of an ordered pair of chains of mappings, called, say, *lower* and *upper*. [23] The both chains must start and finish at the same nodes but have different intermediate nodes. Declaring the pair to be semi-commutative means that $\mathrm{Dom}(lower_1) \subset \mathrm{Dom}(upper_1)$ and $L(x) \subset U(x)$, where x is an arbitrary element of $\mathrm{Dom}(lower_1)$ and sets $L(x), U(x)$ are the values of compositions of mappings occurring into the chains at the argument x. Close to this predicate is *commutativity*, $[=]^{\mathbf{Set}}$, that can be declared for the same configurations and means equality of the lower and upper value sets. Commutativity predicate is defined in **Set** but does not occur into the sketch S.

Now, semantics of S in the universe **Set** is a mapping $[\![\ ]\!] : \mathsf{S} \to \mathbf{Set}$ sending nodes *Person*, *Man*, *Vehicle*, ... into finite sets $[\![Person]\!], [\![Man]\!], [\![Vehicle]\!]$ and arrows $isA_M, owns, \ldots$ into mappings

$$[\![isA_M]\!] : [\![Man]\!] \to [\![Person]\!], [\![owns]\!] : [\![Person]\!] \to [\![Vehicle]\!], \ldots$$

such that the predicates declared for diagrams in S are satisfied. The latter means that statements

$$[\mathtt{inv}]^{\mathbf{Set}}\{f \mapsto [\![owns]\!], g \mapsto [\![isOwned]\!]\},$$
$$[\mathtt{inv}]^{\mathbf{Set}}\{f \mapsto [\![drives]\!], g \mapsto [\![isDriven]\!]\},$$
$$[\Uparrow]^{\mathbf{Set}}\{f \mapsto [\![isA_M]\!], g \mapsto [\![owns]\!], p \mapsto [\![drives]\!], q \mapsto [\![isAC]\!]\},$$
$$\ldots$$

hold true for $[\![-]\!]$.

A.4. Semantics as a parameter

The mapping formulation of semantics above actually forms a general pattern where semantic universe is a parameter. For example, in function of one's needs, one may consider universes $\mathbf{Set}_{10}$,

[23] Chain of mappings is a sequence $f_1, \ldots, f_n$ such that $target(f_i) = source(f_{i+1})$.

$\mathbf{Set}_{01}$, $\mathbf{Set}_{11}$ of sets and partial single-valued, total multi-valued or total single-valued mappings respectively, or, say, a universe $\mathbf{Varset}(\mathscr{T})$ of variable sets and variable mappings depending on a partial ordered set $\mathscr{T}$ modeling (logical) time.

In general, a sketch S in a signature Σ of diagram predicates can be interpreted by a mapping $[\![_]\!] : \mathsf{S} \to \mathbf{U}$ in an arbitrary universe $\mathbf{U}$ having (i) *objects* and *object morphisms* (think of mappings, or transformations, references, interactions,...) between objects) and (ii) Σ-predicates defined. The latter means that for any predicate label $P \in \Sigma$ there is a diagram predicate $P^{\mathbf{U}}$ over $\mathbf{U}$, that is, for any diagram D in $\mathbf{U}$ of the P's arity shape, the statement $P^{\mathbf{U}}(D)$ is either true or false. Now, each diagram D in S (which is actually a mapping as explained above in Appendix A.1), by composing it with $[\![_]\!]$ *becomes* a diagram $D \triangleright [\![_]\!]$ in $\mathbf{U}$. Then each *marked* diagram $P(D)$ in S, that is, a pair (P, D), becomes a predicate declaration $P^{\mathbf{U}}(D \triangleright [\![_]\!])$ in $\mathbf{U}$, which has a certain truth value. An interpretation $[\![_]\!]$ of S is called *valid* if all marked diagrams occurring into S become true statements in $\mathbf{U}$. In conceptual modeling such interpretation are called (valid) *instances* of S.

The pattern just described, keeping the same syntax with varying semantics, reveals a great exibility of the sketch language. As just one example, consider the task of sketch model management (Appendix B.4). The language is syntactically the same but the intended semantic universe $\mathbf{U}$ is now a collection of conceptual models specified by sketches: $\mathbf{U}$'s nodes are sketches, $\mathbf{U}$'s arrows are mappings of sketches and $\mathbf{U}$'s predicate and operations are predicate and operations over sketches. Examples of such operations are sketch merging or integration w.r.t. a certain correspondence information specified by an arrow span (relation) between sketches.

Appendix B. Possible applications of the framework

The framework we described in the paper opens wide possibilities for applying precise mathematical methods to a range of software engineering problems. We will present a very rough and brief outline of possible applications.

B.1. Semantic database theory

A central issue of any consistent database theory is about representation of real world object by values computer can manipulate. In our framework such a representation can be precisely described as a mapping between similar mathematical structures $r : \mathscr{W} \to \mathscr{C}$ with $\mathscr{W}$ being a quasi-topos of variable sets of objects with attributes taking their values in a system of domains $\mathscr{D}$ and $\mathscr{C}$ a quasi-topos of relations over $\mathscr{D}$. Of course, r must be compatible with the structure in its source and target, in particular, with operations $\mathscr{W}$ and $\mathscr{C}$ carry, that is, queries. Elaboration of this approach was started in [12] where a natural transformation of sketches with intended $\mathscr{W}$-semantics into relational database schemas was built.

Moreover, as soon as our framework will be completed with state spaces and object behavior, a task of semantic programming could be stated. What we mean here is designing a set of standard constructs that might be entered into programming languages to implement semantic relationships between object classes (like aggregation, composition etc.) similarly to how *isA*-relationship is implemented via inheritance.

B.2. Query languages in the sketch framework

When data are specified by sketches, operations (queries) are specified by diagram operations over sketches. Precise formal definitions can be found in [10] but the issue of query language's completeness with respect to one or other criteria was not even touched there. A large body of work was done in CT w.r.t. to the issue and, as we mentioned above, a well-elaborated hierarchy of query languages (logical doctrines) was built. However, the CT-context is essentially different and many natural questions, e.g., the place of SQL in this hierarchy, were never posed and need non-trivial research.

B.3. Formal theory of distributed databases

When the same data are represented in multiple data sources (databases), we deal with a "space" version of the object identity phenomenon we considered above in the time dimension. The same object has different states (changeability in time) and different representations in multiple data-sources (changeability in space). Then object identity is explicated by setting space-*same*-relations and time-*same*-relations between extensions of the object class in question in different time moment and different databases; these relations must be compatible between themselves in a certain way. In a bit more detail, if C is a class of real world objects represented in multiple data sources $DB_1, \ldots, DB_n$, then for any moment t we have its extensions $[\![C_1]\!]^t, \ldots, [\![C_n]\!]^t$ connected by space *same*-relations across DB_i's: $[\![C]\!]^t_{1,\ldots,n} \subset [\![C_1]\!]^t \times \cdots \times [\![C_n]\!]^t$. For another moment u, we have $[\![C]\!]^u_{1,\ldots,n} \subset [\![C_1]\!]^u \times \cdots \times [\![C_n]\!]^u$ and for any $t \leqslant u$ there are evident commutativity conditions for space-*same*-relations and local time-*same*-relations ${}^t[\![C_i]\!]^u \subset [\![C_i]\!]^t \times [\![C_i]\!]^u$, $i = 1, \ldots, n$.

A special problem of data distribution is a possibility of conflicts between data representations, especially those usually called semantic conflicts. However, as shown in [6], in the abstract sketch syntax for conceptual modeling all conflicts between different views/representations of the same data are reduced to the only one: data considered basic in one view may be well considered derived in another view and vice versa. Then, in the expressions above for some view i, C_i may stay for the result of some operation O over other classes considered basic in i, $C_i = O(D'_i, D''_i, \ldots, D^{(k)}_i)$. On this basis a consistent and mathematically justified theory of distributed databases could be developed.

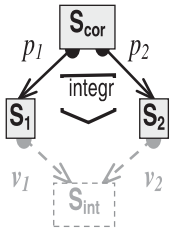
B.4. Model management

The key to a consistent theory of model management is the notion of *model mapping*—it was convincingly demonstrated in [4], see also [11] where views and refinements of conceptual models are treated as mappings between them. A great advantage of sketches in this respect is that there is a very natural and simple notion of sketch mapping: it is a graph mapping compatible with marked diagrams. Moreover, semantics of sketch S in a universe $\mathscr{U}$ can be also considered as a mapping of S into $\mathscr{U}$ arranged as a sketch $\mathbf{U}$ (see Appendix A.3).

In particular, it allows us to present view and refinement mechanisms by simple composition of mappings. A view to a sketch S is set by another sketch (schema) V and a sketch mapping $v : \mathsf{V} \to \mathsf{S}$. Then any semantic instance $[\![\ ]\!] : \mathsf{S} \to \mathbf{U}$ of sketch S immediately gives rise to the corresponding

instance of the view, $v \triangleright [\![-]\!] : \mathsf{V} \to \mathbf{U}$; similar considerations apply to refinements (see [11] for details). In general, a collection of conceptual models together with their views and refinements is modeled by a meta-sketch whose nodes denote models and arrows are mappings between them. Operations over models are then nothing but diagram operations over the metasketch.

For example, the operation of schema integration (merging, in the terminology of [5]) is specified as follows. Given two schemas, sketches S_1 and S_2, the correspondence between them is specified by some schema S_{cor} together with two projection arrows $p_i : \mathsf{S}_{\text{cor}} \to \mathsf{S}_i$, $i = 1, 2$. This is nothing but an application of the relation-via-arrows pattern (Section 2.3.1) in the model management context (see [6] for details). So, the input configuration for schema integration is a sketch span $(\mathsf{S}_{\text{cor}}, p_1, p_2)$ as shown on the diagram below. On the other hand, such a configuration is the input for a diagram operation, the so called *push-out*, well-studied in CT.



Intuitively, applying push-out to an input span integrates the input objects S_1, S_2 modulo the correspondence between them specified by the span; the result is a co-span $(\mathsf{S}_{\text{int}}, v_1, v_2)$ as shown on the diagram above. Particularly, if schemas are just plain sets and S_{cor} is their intersection with inclusions p_i, then applying push-out will produce nothing but union of sets: $\mathsf{S}_{\text{int}} = \mathsf{S}_1 \cup \mathsf{S}_2$ and v_i are the corresponding inclusions. The great advantage of CT is that it allows defining "union" of objects for collections of objects and their morphisms different from sets, particularly, for sketches and their mappings. Thus, schema integration is nothing but an instance of push-out operation which is well-known to be associative.

References

[1] S. Abiteboul, R. Hull, IFO: a formal semantic database model, ACM TODS 12 (4) (1987) 525–565.
[2] M. Barr, C. Wells, Category theory for computing science, Prentice Hall International Series in Computer Science, 1995.
[3] C. Beeri, A formal approach to object-oriented databases, Data & Knowledge Engineering 5 (1990) 353–382.
[4] P. Bernstein, A. Halevy, R. Pottinger, A vision for management of complex models, SIGMOD Record 29 (4) (2000) 55–63.
[5] P. Buneman, S. Davidson, A. Kosky, Theoretical aspects of schema merging, in: A. Pirotte, C. Delobel, G. Gottlob (Eds.), Advances in Database Technology—EDBT'92, Third International Conference on Extending Database Technology, Vienna, Austria, March 23–27, 1992, Proceedings, Lecture Notes in Computer Science, vol. 580, Springer, 1992.
[6] B. Cadish, Z. Diskin, Heterogenious view integration via sketches and equations, in: Z.W. Ras, M. Michalewicz (Eds.), Foundations of Intelligent Systems, Nineth International Symposium, ISMIS'96, Springer Lecture Notes in AI, vol. 1079, 1996, pp. 603–612.
[7] P.P. Chen, The entity-relationship model—towards a unified view of data, ACM Transactions on Database Systems 1 (1) (1976) 9–36.

[8] E.F. Codd, Extending the database relational model to capture more meaning, ACM Transactions on Database Systems 4 (4) (1979) 397–434.
[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. Available from <ftp://ftp.fis.lv/pub/diskin/FISReport 96-03.ps>.
[10] Z. Diskin, Generalized sketches as an algebraic graph-based framework for semantic modeling and database design. Technical Report 9701, University of Latvia, 1997. Available from <ftp://ftp.fis.lv/pub/diskin/ULReport M97.ps>.
[11] Z. Diskin, The arrow logic of meta-specifications: a formalized graph-based framework for structuring schema repositories, in: H. Kilov, B. Rumpe, I. Simmonds (Eds.), Seventh OOPSLA Workshop on Behavioral Semantics of OO Business and System Specifications, OOPSLA98, TUM-I9820, Technische Universitaet Muenchen, 1998. Available from <ftp://ftp.fis.lv/pub/diskin/OOPSLA98.ps>.
[12] Z. Diskin, Representation of objects by values: towards semantic foundations of object-oriented relational database systems. Technical Report 9802, Frame Inform Systems, Riga, Latvia, 1998. Available from <ftp://ftp.fis.lv/pub/diskin/FIS Report 98-02.ps>.
[13] Z. Diskin, Dynamic formal semantics and abstract syntax for the static part of the UML. Manuscript, 2001.
[14] Z. Diskin, On modeling, mathematics, category theory and RM-ODP, in: J. Cordeiro, H. Kilov (Eds.), WOODPECKER'2001: First International Workshop on Open Distributed Processing: Enterprise, Computation, Knowledge, Engineeiring and Realization, (In conjunction with ICEIS'2001), Setubal, Portugal, July 2001, ICEIS Press, Setubal, Portugal, 2001, pp. 38–54. Available from <ftp://ftp.fis.lv/pub/diskin/RMODP01.ps>.
[15] Visualization vs. specification in diagrammatic notations: a case study with the UML (extended abstract), in: M. Hegarty, B. Meyer, H. Narayanan (Eds.), Diagrammatic Representation and Inference, 2nd Int. Conf. on the theory and applications of diagrams, Springer Lect. Notes in AI, vol. 2317, 2002, pp. 112–115, full version is on <ftp://ftp.fis.lv/pub/diskin/Diagrams02.ps>.
[16] Z. Diskin, B. Kadish, A graphical yet formalized framework for specifying view systems, in: First East-Europian Symposium on Advances in Databases and Information Systems, ADBIS'97. St.-Petersburg, September 2–5, 1997, Regular Papers, vol. 1, Nevsky Dialect, 1997, pp. 123–132. Available from <ftp://ftp.fis.lv/pub/diskin/ADBIS97.ps>.
[17] Z. Diskin, B. Kadish, F. Piessens, M. Johnson, Universal arrow foundations for visual modeling, in: M. Anderson, P. Cheng, V. Haarslev (Eds.), Diagrams'2000: First International Conference on the Theory and Applications of Diagrams, Springer Lecture Notes in AI, vol. 1889, 2000, pp. 345–360.
[18] S. Ghilardi, G. Meloni, Relational and partial variable sets and basic predicate logic, Journal of Symbolic Logic 61 (1996) 843–872.
[19] J. Goguen, G. Roşu, Hiding more of hidden algebra, in: Formal methods, FM'99, Springer Lecture Notes in Computer Science, vol. 1709, 1999, pp. 1704–1719.
[20] J. Goguen, J. Thatcher, E. Wagner, An initial algebra approach to the specification, correctness and implementation of abstract data types, in: R. Yeh (Ed.), Current trends in programming methodology, Prentice Hall, 1978, pp. 80–149.
[21] E. Gradel, Ju. Gurevich, Metafinite model theory, in: D. Leivant (Ed.), Logic and computational complexity, Springer, 1995, pp. 313–366.
[22] J. Gray, The category of sketches as a model for algebraic semantics, in: J. Gray, A. Scedrov (Eds.), Categories in computer science and logic, Contemporary Mathematics, vol. 92, American Mathematical Society, 1989.
[23] M. Gyssens, J. Paredaens, D. Van Gutch, A graph-oriented object database model, in: 1990 ACM Symposium on Principles of Database Systems, 1990, pp. 417–424.
[24] M. Hammer, D. McLeod, Database description with SDM: A semantic database model, ACM Transactions on Database Systems 6 (3) (1981) 351–386.
[25] R. Hull, R. King, Semantic database modeling: Survey, applications and research issues, ACM Computing Surveys 19 (3) (1987) 201–260.
[26] IBM. OCL Document Set, 1997. Available from <http://www-4.ibm.com/software/ad/library/standards/ocl.html>.
[27] B. Jacobs, J. Rutten, A tutorial on (Co)algebras and (Co)induction, EATCS Bulletin 62 (1997) 222–259.

[28] B. Jacobs, Coalgebraic reasoning about classes in object-oriented languages, in: H. Reichel, B. Jacobs, L. Moss, J. Rutten (Eds.), Electronic notes in theoretical computer science, vol. 11, Elsevier Science Publishers, 1998.
[29] M. Johnson, C.N.G. Dampney, On the value of commutative diagrams in information modeling, in: Algebraic Methodology and Software Technology, AMAST'93. Springer, 1993.
[30] M. Johnson, R. Rosebrugh, R. Wood, Entity-relationship-attribute designs and sketches, Theory and Applications of Categories, 10 (3) (2002) 94–112.
[31] S.N. Khoshafian, G.P. Copeland, Object identity, in: ACM Conference OOPSLA'86, 1986, pp. 406–415.
[32] M. Kifer, G. Lausen, J. Wu, Logical foundations of object-oriented and frame-based languages, Journal ACM 42 (4) (1995) 741–843.
[33] H. Kilov, Business specifications: the key to successful software engineering, Prentice Hall PTR, 1999.
[34] S.K. Lellahi, N. Spyratos, Towards a categorical data model supporting structured objects and inheritance, in: Next Generation Information Systems, Springer Lecture Notes in Computer Science, vol. 504, Kiev, 1990.
[35] J. Paredaens, P. Peelman, L. Tanca, G-log: a graph-based query language, TKDE 7 (3) (1995) 436–453.
[36] F. Piessens, E. Steegmans, Proving semantical equivalence of data specifications, Journal of Pure and Applied Algebra (116) (1997) 291–322.
[37] J. Rumbaugh, I. Jacobson, G. Booch, The unified modeling language reference manual, Addison-Wesley, 1999.
[38] D. Shipman, The functional data model and the data language DAPLEX, ACM TODS 6 (1) (1981) 140–173.
[39] T.J. Teorey, D. Yang, J.P. Fry, A logical design methodology for relational databases using the extended entity-relationship model, ACM Computing Surveys 18 (2) (1986) 22–197.
[40] A. ter Hofstede, E. Lippe, Th. Van der Weide, Applications of a categorical framework for conceptual data modelling, Acta Informatica 34 (12) (1997) 927–963.
[41] D.C. Tsichritzis, F.H. Lochovsky, Data models, Prentice-Hall, Englewood Cliffs, NY, 1982.
[42] C. Tuijn, M. Gyssens, CGOOD, a categorical graph-oriented object data model, Theoretical Computer Science (160) (1996) 217–239.
[43] M. Wirsing, Algebaric specifications, in: J. van Leeuwen (Ed.), Handbook of theoretical computer science, vol. B, Elsevier/MIT Press, 1990, pp. 673–788.



Zinovy Diskin is a system analyst at *Shulman & Kaufman*, Inc., Detroit, USA, and Research Fellow at *Frame Inform System* (*F.I.S.*), Riga, Latvia. His current research interests are in graphic specification languages and diagrammatic notations, abstract algebraic logic and categorical logic, and their applications in semantic data modeling and object-oriented visual modeling and model management.He got M.Sc. in Mechanical Engineering and Research from Bryansk Institute of Transport Engineering, Russia, in 1981, Ph.D. in Mathematics from the Omsk State University, Russia, in 1992 and Dr. Math. from the University of Latvia in 1994. In 1992–1997 he headed the Laboratory for Database Design at F.I.S.



Dr. Boris Kadish is a founder and the Chairman of *F.I.S. Group*, Latvia. He is also a founder of *ZMarket*, Inc., USA, the leading company in the B2B space in the US market, which is delivering B2B solutions for dozens of large corporations and innovative start-ups including *Toshiba Electronics*, *A&L Distributing*, *MARTA*, *Monster Cable* and others. Dr. Boris Kadish has been an executive in the high tech industry for more than 15 years, always on the leading edge in operations research and computer sciences. Dr. Boris Kadish is listed in the Dictionary of International Biography "Who is Who of Professionals" and in the Dictionary of American Biography "Who is Who of Executives". Since 1991, he has been the scientific advisor of the project "Integration Methods and Tools for Heterogeneous Database Design" performed at F.I.S. and supported by the Science Council of Latvia. Dr. Kadish's high-tech business has included several joint projects with different high-tech companies worldwide. Dr. Kadish has earned a Doctorate degree in Database Design from Moscow Institute of Information Technology, and a Masters degree from Riga Technological University.