Humans, Computers, Specifications: The Arrow Logic of Information System Engineering

Zinovy Diskin[1], Boris Kadish[1]
Lab. for Database Design,
Frame Inform Systems, Ltd.
Riga, Latvia
www.fis.lv/english/science zdiskin@acm.org

Frank Piessens[2]
Dept. of Computer Science,
Katholieke Universiteit Leuven
Heverlee, Belgium
Frank.Piessens@cs.kuleuven.ac.be

Abstract. The goal of the paper is to manifest a special *arrow diagram logic* developed in mathematical category theory as capable to provide a general specification framework for information system engineering. We show that, unexpectedly, abstract ideas developed in categorical logic are of extremely high relevance for approaching some difficult specification problems in the field. Correspondingly, the *arrow thinking* underlying the diagram logic is suggested as a working way of thinking in information system engineering.

Keywords: Semantic Specification, Arrow Logic, Category Theory

1 Introduction: The Problem of Specifying

All the activity of information system engineering (ISE) is saturated with specifications: their design, building, using, reengineering and implementation. Specifications appear, and at once begin to play a key role, at the very beginning when (system) requirements[3] have to be figured out, and continue to play the role up to the end when a few millions of programming code lines together with a few volumes of user's manual are finished. In addition, specifications are extremely important in maintenance of ISs, especially when the maintaining team did not take part in the IS design, as it often happens.

In a sense, the entire design process can be presented by a schematic diagram on Fig. 1. The horizontal branch is a refinement of requirements to programming code through a business/enterprise specification often called *conceptual*, or *semantic, model*, and then through software independent *logical schema* that, finally, is implemented within certain software and hardware. In addition, each node in the way is not a single one-piece huge specification but rather a system of mutually connected component specifications presenting different views on the universe and with different degrees of elaborating details.

[1] Supported by Grants 93.315 and 96.0316 from the Latvian Council of Science

[2] Postdoctoral Fellow of the Belgian National Fund for Scientific Research (N.F.W.O.)

[3] properties that a system should have in order to succeed in the environment where it will be used (Goguen, 1994)

International Journal of Computing Anticipatory Systems, Volume 3, 1999
Edited by D. M. Dubois, CHAOS, Liège, Belgium, ISSN 1373-5411 ISBN 2-9600179-4-3

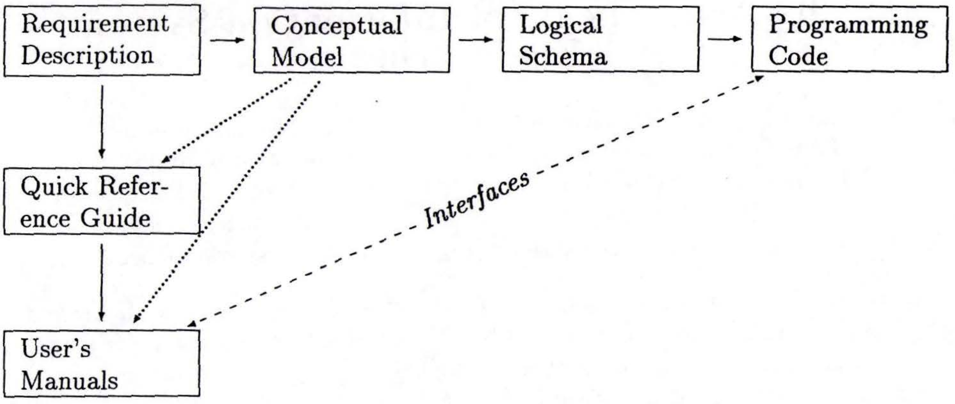


Fig. 1: Schematic view on IS design

The vertical branch is a refinement of fuzzy expectations of what is desired to quite concrete directions of what is to do.

The focus of the present paper is on the top-left "generating" corner of the diagram centered (conceptually) around specifying requirements and building semantic specifications, the corresponding activities are called *requirement enginering* (RE) and *conceptual/semantic modeling and design* (CMD). In essence, these activities are complex processes of communication between software designers, experts on the universe-of-discourse and users of the future IS, which should result in a well organized system of compact comprehensible specifications of the universe and the user's expectations of the system.

The key peculiarity of the process is that the specifications in question have to be somehow extracted from multiple, very informal and often hardly consistent descriptions provided by different people. Moreover, as the design is going on, and mutual understanding between the contributors is increasing, these partial descriptions may be essentially changed so that the specifications are permanently revised and corrected. On a whole, the design process appears as a complex interplay of anticipating "Why", specifying "What" and implementing "How" and combines in a non-trivial way logical, cognitive, technological and humanitarian (psychological and social) aspects. It necessarily goes along some specific spiral rather than a direct line, and the entire specification structure, and its components as well, are endlessly reengineered until the approaching deadline captures them as they are.

1.1 Specifications in Conceptual Modeling and Design

It is clear from the above, and it was stressed and displayed by Goguen with coauthors (see (Goguen, 1996) and references therein), that social contexts are very important in RE and remain valid in CMD too, thus a certain informality in high-level specifications is unavoidable and even useful. On the other hand, effective communication assumes a certain degree of common understanding of the domain in question, which should be fixed in a sufficiently precise domain specification; this is especially important for business/enterprise specifications, that is, their conceptual models.

A conceptual model must record the information discovered in RE in some precise form, in addition, it should be flexible and open for reengineering. The task is highly non-trivial since a typical universe to be specified is not an example from a textbook but a piece of reality like a bank or an airport. One may imagine an elementary verbally formulated arithmetic problem that is very simple in essence but whose formulation takes hundreds of pages. The problem is not in resolving algorithm as such but in specifying a huge array of data in a comprehensible way, we again remind and stress that the very data are subjected to discussion and permanent changes. In fact, such problems are out of scope of classical science: normally, the latter deals with either continuous domains specified by differential equations, or homogenous discrete domains specified by, say, systems of linear equations.

In contrast, an airport or bank are discrete heterogeneous systems consisting of many subsystems of different kinds, which, in their own turn, consist of many objects of different kinds, all are integrated in a single whole via complex mutual relationships and mutually dependent functioning subjected to dozens of specific (business) rules. Importantly, each subsystem can well induce its own specification language to be effectively used in communication between designers and domain experts, thus, the entire conceptual model appears as a heterogeneous specification structure. Moreover, though in ideal semantic specifications should be entirely independent on future implementation, in practice the intended implementation influences the choice of semantic languages and conceptual models become even more heterogeneous because of extreme heterogeneity of the modern software.

So, a conceptual model has to compress a bulky array of diverse information into a compact specification suitable for human communication. A natural, and the only practical, choice is to use graphic languages, and indeed, a vast variety of graphic notational systems was developed in RE and CMD. On the other hand, a model should be sufficiently precise to rule out ambiguities and be readily transformed into formal software specifications in the further stages of design. Thus, a good semantic modeling language should be graphic and comprehensible, sufficiently precise or even ready to be formalized, and sufficiently expressive to capture all the peculiarities of the real world. In addition, the language should be flexible to manage the heterogeneity problem within a single unifying pattern.

Up to the current state of the art, this synthesis has not been achieved: graphic specification languages in use are either far from being precise, or have a very limited expressive power, or both. In addition, they are based on very different and particular (if any) logical foundations so that the entire area appears today as somewhat like a Babylon of graphic notational systems. There is even the Tower of Babylon - it is the so-called Unified Modeling Language (UML) recently adopted by a consortium of software vendors as an industrial standard in the field of conceptual modeling and design (OMG, 1997). No doubts, UML is a significant achievement towards unification but, from the mathematical view point, it is just a one more (monstrous) notational system rather than a universal framework capable to unify the diversity of RE/CMD-notations. Indeed, since the UML constructs have no formal semantics, one has no precise means to relate and compare one's particular model with UML. Moreover, by the same reason, an UML specification can be differently treated by different users and so, solving some technological problems the UML creates new ones.

1.2 Specifications in Software Engineering[4]

Modern (and of the nearest future) ISs are cooperative and heterogeneous: they appear as a large number of local ISs distributed over complex computer/communication network (intranet). In its own turn, every local IS is a complex configuration of data flows between data processing units, and the pattern of global/local is reiterated downward and upward.

Normally, local ISs are sufficiently autonomous and can significantly differ in their origin and development, hence, in their organization, architecture, data models *etc.* A result is that inside of intranet different nodes are often operated by different software which runs on different hardware platforms. All this constitutes the phenomenon of *architectural heterogeneity*. In addition, data circulating through the network significantly differ in their meaning (which can range from natural numbers to video images) and their types (`Integer`, `String`, `Figure` *etc*); this constitutes *semantic heterogeneity*. Builders of complex software systems are thus faced with the need to integrate heterogeneous data residing in different sources over some communication network. Today the problem is becoming global in both direct and figurative senses: with the advent of the information superhighway, in addition to the many intranet enterprise databases, there is now a vast amount of information (data and software) accessible through the Internet.

Handling architectural heterogeneity is usually attributed to the so called *interoperability* and amounts to resolving communication problems (file transfers, remote logins *etc*). It can be stated that by now the basic conditions for interoperability in heterogeneous systems either have been achieved or will be achieved in the nearest future (Drew et al., 1993). In contrast, managing semantic heterogeneity, *ie*, *interoperation*, is far from being solved and, moreover, as is noted in (Drew et al., 1993), *the problem itself is still at the stage of being understood* (and hence, we add, requires a corresponding logic to reason properly).

Clearly, the interoperation problem is a specification problem: to manage it one should be able to specify heterogeneous data structures in precise and unifying abstract terms so that specifications (*what*) would be separated from algorithms (*how*). Specification of data processing unit is like a plug for an electrical appliance. Ex-

[4] Some terminological remarks seems to be useful. The term "software engineering" (SE) is sometimes used in a wide sense covering all the schema on Fig. 1 above, and sometimes more narrowly focusing more on the right half of the refinement chain. We will try to keep using the term "SE" more in the narrow sense and use "ISE" for SE in the wide sense; at the same time we remind that in this paper we focus on the high-level - semantic - part of ISE-specifications, somewhat like "proper" ISE.

ternally, the latter is completely characterized by a few numbers (voltage, power, current frequency) so that to connect appliances between themselves or to a network all that is required is to verify matching of several numbers. Of course, we cannot hope on such a simple story for data processing devices but the network of electrical appliances is a sample useful to have in mind (we borrow this metaphor from (Makowsky, 1992)).

Unfortunately, the picture one can observe in the modern software is in a sharp contrast with this sample: specifications are often implicit and hidden in the implementation so that a procedural description is the statement of purpose and its resolving algorithm simultaneously (hence, any discussion of success and optimality of the algorithm is canceled). On a whole, while all the prerequisites of handling interoperability are achieved, (semantic) interoperation is still a challenge.

To summarize, extreme heterogeneity of the software world has led to extreme heterogeneity of software descriptions. The latter are often overly sugared syntactically and overloaded with particular details of presentation and implementation. The role of semantic specifications in such a heterogeneous and notationally diverse environment becomes much more important and even crucial. The area strongly needs an integral framework of powerful specification principles capable to cover different data and their semantics in a uniform way. No such a framework is known to the community and its specification-linguistic efforts have resulted in a huge diversity of *ad hoc* notational systems and languages – the Babylon metaphor, again, was used by different people in their attempts to characterize the situation.

1.3 What We Suggest

Since any language, and even merely a notational system, is a more or less direct reflection of the corresponding underlying logic, software engineering is actually suffering badly from the lack of suitable logics. Thus, the problem is in specificational logics rather than in notational tricks – this consideration is crucial for the field but often is not well understood.

The goal of the present paper is to manifest a special *diagram* or *arrow logic* developed in mathematical category theory as capable to solve a lot of specification problems in IS-engineering. We will demonstrate effectivity of arrow logic in approaching only two but fairly infamous problems: heterogeneity of conceptual modeling languages and managing specification repositories. As for other specification-centered problems in ISE, an optimistic forecast will be briefly motivated in section 3.5 .

We did not invent the arrow logic ourselves. Surprisingly, but a graphic yet quite precise specification paradigm extremely suitable for ISE applications has already been invented in a quite abstract branch of modern algebra, category theory (CT), where an approach to specifying mathematical structures via the so called *sketches* was developed. Being adapted for ISE-needs, sketches become an extremely expressive, flexible and handy specification means (cf.(Johnson and Dampney, 1993; Diskin, 1997a; Piessens and Steegmans, 1997; Cadish and Diskin, 1996; Diskin, 1998b; Diskin, 1998a)). Moreover, an important observation we made while applying sketches is that in a concrete domain sketches normally appear as a precise formal refinement of the existing notation rather than an external imposition upon the domain. In particular, sketches can be seen as a far reaching generalization of entity-relationship diagrams in conceptual modeling, interaction diagrams in process modeling, schema grids in structuring specification repositories.

In the rest of the paper the essence of the arrow logic is briefly discussed and demonstrated in a few simple examples. Some more general yet speculative suggestions are made in section 4.

Acknowledgements. We are indebted to the participants of our sketch seminar at F.I.S.: Sergey Ageshin, Ilya Beylin, Vitaly Dzitenov and Felix Skorohod for numerous fruitful discussions and stimulating criticism; as for this latter component, Ilya Beylin should be specially thanked for often sharp but always substantial and really stimulating discussions. Several substantial conversations on the subject with George Sheinkman also were extremely helpful. Special thanks go to Vitaly Dzitenov – once more – for implementation of a prototypical sketch editor and real using sketches in this software project.

Finally, we are grateful to Daniel Dubois who (unexpectedly for us) invited us to prepare this paper for CASYS, and for encouragement we had been got from him during preparation.

2 Ideal Specification Paradigm for Software Engineering

The bundle of specification problems outlined in introduction can be resolved only within a framework of powerful *specification paradigm* rather than a single extra-universal specification language. We mean that one needs a generic protolanguage $\mathcal{L}(\mathbf{p})$, whose definition depends on a list of parameters $\mathbf{p} = (p_1, \ldots, p_n)$[5] so that different specification languages would be instances of the protolanguage defined by the choice of parameters: given a problem domain A, a specification language L_A suitable for A may be built as $\mathcal{L}(\mathbf{a})$ for some $\mathbf{p} = \mathbf{a}$ appropriate for A. Another domain B needs another language $L_B = \mathcal{L}(\mathbf{b})$ and so on but the *diversity* of domains $A, B, \ldots$ results in a *variety* of languages $L_A, L_B, \ldots$ since all these languages are treated in a uniform way as $\mathcal{L}$-instances.

To achieve its goals, the paradigm $\mathcal{L}$ should possess the following characteristics:

Semantic capabilities. $\mathcal{L}$-languages should admit semantic interpretation close to the real world semantics, that is, at least, be object-oriented and capable to specify object class schemas in a natural way. On the other hand, $\mathcal{L}$-languages should admit value-oriented interpretations and be capable to specify type schemas.

The semantics of the $\mathcal{L}$-languages should be *formalizable* to rule out ambiguities, and to allow for computer assisted verification.

[5] of course, p_i are not numbers but formal constructs which can themselves be complex structures

Universality. A wide range of semantic constraints (business rules) should be expressible. Moreover, the totality of semantic constraints possible in the real world is practically unlimited, and even given some particular case, the set of business rules is usually changeable and subjected to unpredictable evolution. So, universality of specification language must be understood in the absolute sense: *any* possible (formalizable) semantic constraint must be expressible.

Unifying flexibility. An *ad hoc* specification language (say, L) developed for a particular domain can accumulate a useful experience and be convenient and customary for many experts in the domain. It would not be reasonable to neglect these advantages of L and replace it with some equally expressive but entirely novel $\mathcal{L}$-language. A better solution would be to simulate L in $\mathcal{L}$, that is, to find parameters $\mathbf{p} = \mathbf{a}$ such that syntax and intended semantics of $\mathcal{L}(\mathbf{a})$ would be close to that of L. Thus, $\mathcal{L}$ should possess the possibility to simulate a wide range of *ad hoc* languages.

Abstraction flexibility. This term means the capability to build specifications in a data model independent way. Consider, for example, the notion of *view* to data. What is a view is quite clear in the case of the relational data model but the notion is of extreme importance far beyond it: views make software applications tolerant to changes in the data schema. Thus, one needs a general definition applicable to the relational, OO, semantic *etc* data schemas in a uniform way. In other words, one needs an abstract definition of view **where a data model would be a parameter**. The same holds for other basic metadata modeling concepts such as schema refinement or schema integration. That is, one needs an integral metadata modeling framework where a data model would be a parameter. We call the possibility to state such a framework *abstraction flexibility* of the specification paradigm.

Comprehensibility. Any specification language, as though powerful and flexible it would be, will remain a thing-for-itself if it is heavily comprehended by the human. We recognize at least two main components of comprehensibility: graph-based evidence and modularizability.

(i) Graphical syntax. People like to draw graphical schemas to facilitate reasoning and communication. Usually these schemas are considered as informal heuristic pictures, while to become precise and implementable they must be converted into string-based specifications similar to theories in logical calculi. What would be desirable in this respect is to build a graph-based specification language such that graphical images themselves should be precise specifications suitable for implementation.

(ii) Modularizability. There is a growing awareness among experts that most specification and verification methods have reached their limits but can handle only systems of small size and complexity. As it was noted in (Langmaack et al., 1997), "if there is a hope that industrial-size designs can be handled by formalized methods, it must be based on the premises of compositionality and

abstraction". If so, the "ideal" paradigm $\mathcal{L}$ we are discussing should necessary include a flexible modularization mechanism. That is, there must exist a machinery of decomposing a global L-specification S into simpler component L_i-specifications $S_i, (i = 1, \ldots, n)$ where L, L_i are $\mathcal{L}$-languages, such that an IS satisfying all the S_i must automatically satisfy S or, in another context, component systems IS$_i$ satisfying S_i can be then integrated into the global system IS satisfying S. Of course, the schema of decomposing S into S_i should be presentable in a concise form (preferably graphical).

Specification methods currently used in software engineering fail to integrate the requirements stated above in a single framework. Moreover, with current specification methodologies the task of building the "ideal" paradigm we have discussed appears to be extremely difficult if at all possible. In particular, concerning the property of abstraction flexibility, it seems impossible to speak about data schemas and data instances in the abstract way without any specific description of what they are: this appears to be a kind of substantial speaking about nothing (cf. (Drew et al., 1993)).

Fortunately, as we have said, a methodology and specification machinery ideally matching the ideal properties we have formulated, have been developed in mathematical category theory. In the next section we will describe briefly main lines of incorporating categorical means into the real specification problems of semantic data modeling and structuring schema repositories.

3 Arrow Logic of Software Specifications

3.1 Arrow Thinking and Category Theory

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

The basic idea underlying the approach consists in specifying any universe of discourse as a collection of *objects* and their *morphisms* which normally are, in function of context, mappings, or references, or transformations or the like between objects. As a result, the universe is specified by a directed graph whose nodes are objects and arrows are morphisms.

Objects have no internal structure: everything one wishes to say about them one has to say in terms of arrows. This feature of CT can be formulated in the OO programming terms as that object structure and behaviour are encapsulated

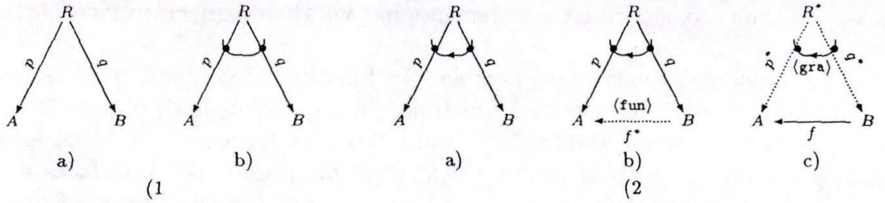


Fig. 2: Relations via arrows

and accessible only through the arrow interface. Thus, objects in the CT sense and objects in the OO sense have much in common.

A surprising result discovered in CT is that the arrow specification language is absolutely expressible: any construction having a formal semantic meaning[6] can be described in the arrow language as well. Moreover, the arrow language is proven to be an extremely powerful conceptual means: if basic object of interest are described by arrows then normally it turned out that many derived objects of interest can be also described by arrows in a quite natural way. The main lesson of CT is thus that to define properly the universe we are going to deal with it is necessary and sufficient to define morphisms (mappings) between object of the universe. In other words, as it was formulated by a founder of categorical logic Bill Lawvere, *to objectify means to mapify.*

For example, a relation between two objects A and B, where A, B can be object classes, or process interfaces, or data schemas is specified in the same way by an arrow span shown on schema (1a) Fig. 2. There R is an object similar to A, B, which is to be thought of as consisting of relationships between items of A, B, and p, q are projection morphisms (references, processes, schema mappings) to be thought of as extracting the first and the second components of relationships. In addition, a special predicate should be declared for the pair (p, q) to ensure that R-items can be indeed thought of as couples composed from A-items and B-items. A natural way to express such a declaration syntactically is to mark the arrow diagram (p, q) by some label, say, an arc, as shown on schema (1b), Fig. 2. This schema is nothing but a simple sketch.

Specifically, the property of being a functional relation, say, from B to A, can be expressed by another predicate declared for the same diagram; syntactically, it can be presented, for example, as shown on sketch (2a), Fig. 2. In this case there is a mapping
$f\colon B \to A$ derived from the relation. In fact, one has a diagram operation producing an arrow from a span marked with the functional relation label, see schema (2b), where the marker ⟨fun⟩ denotes the operation in question and the dotted body of the f^*-arrow denotes that it is derived (by the operation). The converse construction of building the graph of mapping is presented on sketch (2c) where all derived items are marked with the *-superindex.

So, in the arrow framework relations can be specified, and manipulated, in an

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

abstract way without considering their elements but via their arrow interfaces, (p, q) in the example.

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

The arrow specification framework gives rise to a special kind of thinking – the so called *arrow thinking*. In philosophical terms, the arrow thinking can be seen as a precise formalization of the dialectics idea (but, of course, CT as such is a mathematical discipline having nothing common with philosophical speculations). In this context, the arrow thinking has a long history going back to ancient Taoists and continuing by now with the OO paradigm in software and the constructive ontology in quantum physics (see (Diskin and Kadish, 1996) for constructivity aspects of arrow specifications). It would not be a great overstatement to say that CT offers a mathematically justified framework for a proper formalization of the very general ideas of observability, constructivism and object-orientation.

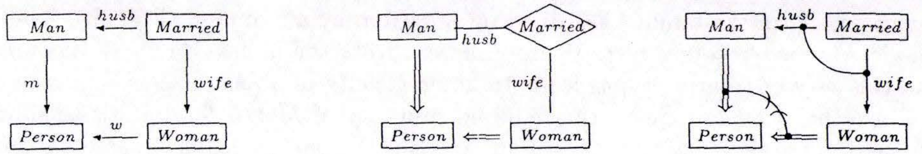
Formally, a category is a directed multigraph with composable arrows: given any two arrows $f\colon A \to B$ and $g\colon B \to C$ where A, B and C are objects, an arrow $h = f \rhd g\colon A \to C$, the *composition* of f with g, is defined in a unique way. Due to arrow composition and, often, other arrow diagram operations, categories modeling complex universes are usually infinite and, hence, implicitly specify also a lot of derived information about the universe. In practice, one deals with finite presentations of infinite categories. These finite presentations are nothing but *sketches* and so sketch specifications appear as a constructive technological realization of the arrow logic (see (Diskin, 1997c) for a database oriented presentation of sketches).

3.2 Graph + Diagram Predicates = Sketch

Arrow specification with diagram predicates is a sketch, and it follows from the general results of categorical logic mentioned above that any formalizable specification can be replaced by a corresponding equivalent sketch. And even more, there is a fixed (and not very big) collection of diagram operations, compositions of which cover all formalizable constructions.

The sketch specificational vocabulary is strict and compact. Sketches are graphical constructs consisting of three kinds of items: (i) nodes, to be interpreted as object (*eg*, sets), (ii) arrows, to be interpreted as morphisms (*eg*, functions), (iii) marked diagrams, i.e., labeled collections of nodes and arrows, to be interpreted as predicates (*eg*, declared for diagrams of sets and functions).

Of course, before one can draw a sketch, a collection of diagram predicates (markers) should be declared and organized into a *signature*, say, Π. That is, any sketch is a Π-sketch for some predefined signature Π.



a) Sets-and-functions graph b) ER-diagram c) Sketch

Fig. 3: Internal object structure via arrow diagrams

It is important that diagram predicates can be defined in a uniform way. Their semantics can be defined via so called *universal properties*, or equivalently, using some essentially algebraic axioms. This way of defining things is very common in CT. It effect, declaring a diagram predicate can be reduced to declaring some equations for corresponding diagram operations. So, any signature of diagram predicates can be considered as an essentially algebraic theory over some predefined set of basic operations.

An important property of sketches is that their framework is sufficiently manageable to allow for correctness proofs of practically and mathematically valid algorithms on sketches. In particular, in (Piessens, 1996; Piessens and Steegmans, 1997), algorithms for deciding a basic problem of semantic equivalence of sketches, and performing their integration as well, are developed and proven correct for sketches of some specific signature.

3.3 Semantic Modeling via Sketches

As it was said above, current semantic models are either hardly comprehensible in the case of complex specifications, or informal, or have a very limited expressive power. In contrast, the arrow framework is as rigor as logical calculi and absolutely expressible yet it is graph-based. The specification principle underlying the approach is that *the world consists of sets and functions.* Correspondingly, the world can be specified by a directed graph whose nodes denote sets (classes of objects) and arrows denote functions (references or attributes).

3.3.1 Internal Structure of Objects via Arrows

The basic discovery of category theory is that it is possible to describe internal structure of objects in a class externally via stating certain properties of arrow diagrams adjoint to the class. Let us consider, for example, the class-reference diagram shown on Fig. 3(a).

We can infer from this diagram that to each object *o* of the class *Married* are assigned some *Man*-object *o.husb* and some *Woman*-object *o.wife*. However, this does not allow to state that each *Married*-object is a pair of *Man*-object and *Woman*-object. To ensure the latter property, in many semantic models class *Married* would be declared to be a relationship class between *Man* and *Woman*, and correspondingly marked, *eg*, in the famous ER-diagram notation, by a diamond (Fig. 3b). Then the node *Married* becomes intrinsically different from the nodes *Man* and *Woman* and so this way of treating types leads to heterogeneity of object classes.

Another way to express the required property of *Married*-objects is to shift the focus from nodes to arrows and to declare that the pair (*wife*,*husb*) has the following *separation property*:

$$\text{for any objects } o, o' \in \mathit{Married} \text{ if } o.\mathit{husb} = o'.\mathit{husb} \text{ and } o.\mathit{wife} = o'.\mathit{wife} \text{ then } o = o'$$

Indeed, in such a case the function $\langle husb, wife\rangle\colon\ Married \to Man \times Woman$ is one-one so that *Married*-objects can be identified with some pairs $(m, w) \in Man \times Woman$. Thus, a property of the arrow diagram (*wife*,*husb*) enforces a certain internal structure of *Married*-objects.

If we introduce a diagram predicate of being *separating* (it is a particular case of the general construct considered in example on Fig. 2), and agree to denote the family satisfying the predicate by an arc, then the required specification of the class *Married* will look as shown on Fig. 3(c). The picture contains two other diagram markers: the double-body of arrow and arc with brackets. The former is a marker that might be hung on diagrams consisting of a single arrow and denotes the property of being IsA-arrow. The arc marker denotes the property of commonly targeted family of functions to cover the target disjointly, that is, each object in the target class is in the image of one and only one source classes. Thus, the specification (c) states that (i) each *Person*-object is either *Man*-object or *Woman*-object but not both and (ii) each *Married*-object is actually a pair $(m, w) \in Man \times Woman$.

The example above is quite simple but it demonstrates how to specify various object types via arrow diagrams and their properties (see (Diskin, 1998b) for other examples). Moreover, it turned out that rich semantic constructs – IsA, IsPartOf, aggregation and qualifications relationships between objects – also can be specified in the framework of sets-and-functions, correspondingly, of arrow diagram predicates. Sets and functions to be considered here should be changeable, that is, variable in (logical rather than physical) time: it was shown in (Diskin and Kadish, 1996) that all semantic constructs above can be exhaustively described in the framework of variable set semantics for sketches. However, variable sets and functions can be specified in the logic of arrow diagram predicates as well as constant ones. Moreover, the construction of variable set universe is well studied in CT under the name of *topos*. In these terms the experience of semantic modeling can be summarized as experience of viewing the real world as a topos of variable sets while building semantic model is nothing but building a finite graphic presentation of this topos – a graph-based analog of well known algebraic task. The topos-theoretic refinement of semantic modeling has far reaching consequences for stating the discipline on firm mathematical foundations. In particular, it opens the door for applying powerful algebraic techniques developed in category theory, specifically, diagram chasing. Unfortunately, discussing this and other aspects of sketching semantic modeling goes far beyond the frame of the present paper.

3.3.2 Sketches vs. Heterogeneity of Semantic Models

The value of the graph-based logic approach to semantic modeling is not exhausted by an adequate formalization of semantic models. Maybe, even a more important benefit is that the approach facilitates managing the infamous problem of heterogeneity of semantic models. Indeed, even a surface glance over the current semantic modeling shows an abundance of various notational systems with a great diversity of conventional graphic constructs, symbols, markers *etc*, which makes comparing and integrating different semantic models an extremely difficult task.

This Babylon mess inspired a lot of theoretical work in academia (like, *eg*, (Atzeni and Torlone, 1996)) and in industry where the famous UML was adopted as a standard in OO analysis and design. However, in the light of mathematical treatment of semantic modeling we have outlined above, these attempts look conceptually helpless and too partial.

We suggest the following way to manage heterogeneity of semantic models. Given a model $\mathcal{M}$, its vocabulary of specificational constructs is specially arranged to become convertible into a signature of diagram predicates, $\Pi_{\mathcal{M}}$, so that $\mathcal{M}$-specifications could be converted into $\Pi_{\mathcal{M}}$-sketches. Moreover, by adjusting visualization of $\Pi_{\mathcal{M}}$-predicates one can make visual presentations of $\Pi_{\mathcal{M}}$-sketches very close to semantic $\mathcal{M}$-schemas as they are seen externally (see (Diskin, 1998b) for details). In this way the diversity of semantic models can be transformed into the variety of sketch data models in different signatures. Indeed, sketches in different signatures are nevertheless sketches, and they can be uniformly compared and integrated via relating/integrating their signatures. A more detailed discussion can be found in (Diskin, 1998b).

The methodology of the sketch approach can be illustrated by the following analogy (invented in (Diskin, 1998b)). In a sense, the place of sketches in the heterogeneous space of semantic models is comparable with that of the modern positional numeral systems in the general space of numeral systems including also zeroless positional systems (*eg*, Babylonian, Mayan) and a huge diversity of non-positional *ad hoc* systems (Egyptian, Ionian, Roman *etc*). The analogy we mean is presented in the table below.

Specification paradigm	Data to be specified	Language		Minimal language
		Predefined base	Specification	
Sketches	Collections of sets and functions	Signature, Π	Π-sketch	Original categorical sketches
Positional numeral systems	Finite cardinalities	Base of numeral system, k	Positional k-number	Binary numbers

In the context of this analogy, many of conventional notations used for conceptual modeling are similar to *ad hoc* non-positional numeral systems like, *eg*, Roman. In particular, operating sketches in different signatures is like operating natural numbers written in different positional systems. The question about expressive power of sketches is analogous to the question of whether a positional numeral system can emulate an arbitrary numeral system. The positive answer is evident to everybody but it would not be so if one considers the question within the pure syntactical frame: thinking syntactically, it is not so obvious how to translate Roman numbers into decimal numbers. The question above is easy because of our inherited habit to think numeral numbers semantically. Indeed, thinking semantically, any Roman number is a presentation of some finite cardinality, and a decimal number can express the latter as well.

The situation with sketches is somewhat similar: thinking semantically, any data schema is a specification of system of sets and functions, and the latter can be expressed by a sketch as well. Of course, a specialist in semantic modeling who has the habit to think of conceptual schemas in pure relational terms will have doubts whether an arbitrary complex logical formula can be expressed by a diagram predicate. The translation is indeed far from being evident but nowadays can be found in any textbook on categorical logic (see, *eg*, (Barr and Wells, 1990)).

3.4 Meta-specification Modeling via Sketches

It is clear that a working description of large, and of moderate size too, real system cannot be a flat, one-piece, specification. Rather, it will be a collection of different viewpoint specifications and their refinements of different degree of elaborating details. In addition, the components (views and refinements) can be somehow additionally related between themselves, for example, one component specification can be the intersection of several others, or the merge of some others, or both. So, specification of complex system is itself a complex structure subjected to certain integrity constraints and carrying certain operations. To fix terminology, we will call component specifications *schemas* and the entire specification repository *schema library*.

To be manageable, a schema library should be structured according to some predefined pattern and so the key to the problem is in finding a language suitable for specifying the library structure. Several proposals for such languages were made in research papers and/or implemented in CASE-tools. It is common to arrange these languages around the structuring primitives of view to a schema, refinement of a schema and schema integration (cf. (Batini et al., 1993)). In particular, in a series of works by the Italian school of conceptual modeling (Batini et al., 1993; Santucci et al., 1993; Francalanci and Pernici, 1994), a general mechanism of *view/refinement grid* was proposed for structuring schema libraries.

However, their grid and similar patterns that can be found in the literature lack a construct of vital importance for the problem: namely, what is a structuring pattern for specifying relations between schemas? In other words, given two conceptual data schemas S_1 and S_2, how can one specify that S_1 is a view on S_2 or, for example, that a third schema S_{Int} is the result of integration of S_1 and S_2 according to some additional information about correspondence between S_1 and S_2. Similarly, how can one specify that a schema S' is a refinement of schema S?

It was shown in (Diskin, 1998a) that basic modularization concepts of view to data and refinement of data can be naturally described by arrows denoting sketch mappings or *functors* (they are graph mappings compatible with diagrams).

A *view* to a schema (Π-sketch) S is a pair $V = (S_V, v)$ with S_V is the *view schema* (another Π-sketch) and $v\colon S_V \to \overline{S}$ is the *view mapping* (functor) into some augmentation of S with derived items.

A *refinement* of a schema S is a pair $R = (S_R, r)$ with S_R is the *refinement schema* and $r\colon S \to \overline{S_R}$ is the *refinement mapping* into some augmentation of S_R with derived items.

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

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

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

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

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

$$\begin{tikzcd}
S_V \arrow[r, "\rho_i^{S_V}"] \arrow[d, "v"'] & \overline{\mathcal{F}_i(S_V)} \arrow[d, "\mathcal{F}_i(v)"] \\
\overline{S} \arrow[r, "\rho_i^S"] & \overline{\mathcal{F}_i(\overline{S})}.
\end{tikzcd}$$

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

Thus, a constraint induced by the software reality leads to a mathematically justified construct. We consider this as an instance of the general phenomenon we have stressed above: extremely high relevance of the arrow language developed in category theory for software engineering.

The construction of hypersketch fibration can be generalized in the following way. Till to now we assumed that the collection of hypersketches is organized in a refinement chain. However, it may happen that the schema repository is a set of such chains outgoing from a common source, for example, these chains can represent different directions of the design project. Moreover, there is nothing strange if these chains meet in some node and then diverse again following to some project schema. In other words, one can well assume that refinement steps are organized into some graph, actually, a sketch, representing the project on a very abstract level, details can be found in (Diskin, 1998a).

3.5 Sketches vs. Software Specifications: a Conjecture

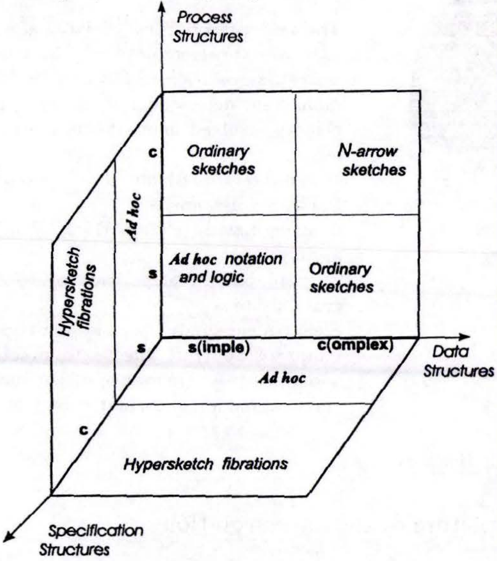
A sketch is a precisely defined formal construct: it is a directed multigraph endowed with diagrams labeled by markers from a predefined signature. So, sketches can be studied by mathematical methods, this is a subject of category theory and the basic framework is already established (Diskin, 1997c).

Modeling software notions and constructs by sketches is a subject of computer science beyond mathematics. It can be roughly divided into data modeling, process modeling and specification (meta-) modeling. It was shown above how data modeling and meta-specification modeling can be managed with sketches. Process modeling can be approached via *n-arrow* sketches, but by lack of space, we refrain from discussing it in the paper

If the structures to be modeled are simple, ordinary *ad hoc* specificational means are sufficient. Complex structures however need special means and our considerations above hopefully show that the sketches is a powerful and natural specification paradigm.

Applicability of the sketch framework can be visualized by Fig. 4 in which some specification space is presented. In modern complex ISs built on OO principals, all the three components - data, procedure and meta structures are complex, and then the structural pattern for specifications in a modern IS is that of *n-arrow hypersketch fibrations* (see Fig. 4a).

The flexibility of sketches as a modeling (correspondingly, specification) tool is provided by existence of different substantial interpretations of sketch items (Fig. 4b). Interpretations 3,4,5 residing in the square [DS=simple, PS=complex] on the plane MS=0, have been thoroughly studied in semantics of computation research, *eg*, (Jifeng and Hoare, 1989; Moggi, 1991; Wadler, 1992). Interpretations 1 and 2 (here we mean data structuring aspects of OO), the square [DS=complex,PS=simple], were developed in (Johnson and Dampney, 1993; Diskin and Kadish, 1995; Diskin and Kadish, 1996). In particular, the sketch-based solution of the long-standing problem of heterogeneous view integration was described in (Cadish and Diskin,



a) Space of software specifications (s – simple, c – complex)

b) Possible interpretations of sketch nodes and arrows

1. sets and functions (in *data modeling*);
2. object classes and references (*OO analysis and design*);
3. data types and procedures (*functional programming*);
4. propositions and proofs (*logic/logic programming*);
5. interfaces and processes (*process modeling*);
6. data states and transactions (*transaction modeling*);
7. schemas and schema mappings (*meta-modeling*).

Fig. 4: IS-engineering specifications in the arrow logic framework

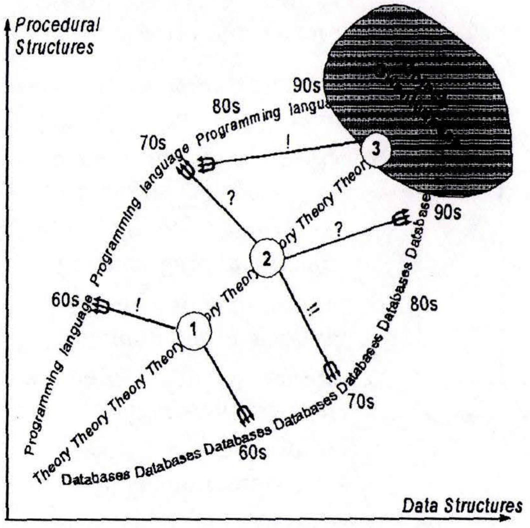
1996)). Modeling transactions, the square [complex,complex], is in the stage of initial development if to speak about technology.

Meta-modeling, the plane PS=0, was developed in (Diskin and Kadish, 1997; Diskin, 1998a); in (Diskin, 1997b) an integral data model independent framework for specifying database architecture was proposed.

4 Instead of Conclusion: Mathematics vs. Information System Engineering – A General Perspective

Any human activity is impossible without descriptions of *why*, *what* and *how*. The necessity of *how* is obvious, *why*-statements are of vital importance when we speak about humans but *what*-descriptions, though important, are not mandatory: the activity is richer than logical schemas and goes beyond denotational formulations. However, as it goes further, the absence of clear specifications becomes a more and more serious obstacle till the benefits of having a specification language will exceed efforts for its development.

Software is computer's intellect somewhat similar to human's one, and simultaneously, software is a human activity which seems has reached the stage when clear semantic specifications become a crucial factor. The question is in languages suitable to achieve the goal. In its turn, semantic specifications are impossible out of abstract mathematical framework, so, the question is in suitable mathematics.



The axis scales richness of data/ procedural structures employed. The curves trace the evolution of DB and PL. The middle line depicts logical/algebraic machinery involved in mathematical models:

1. discrete mathematics / matrices, Boolean polynomials;
2. string-based (relational) logic / ordinary algebras;
3. graph-based (categorical) logic / diagram algebras.

Each trident symbolizes a systematic attempt of applying theory to practice, and its success (relevance of the mathematical model) is evaluated by ! or ?

Fig. 5: An overall picture of software evolution

Of course, there are always genetic gaps between mathematics and applied methodology, and then between methodology and practice, and this is quite normal. What is not normal is the size of these gaps and disagreement between their opposite sides which one can observe in modern SE. This has resulted in a common general disappointment of practitioners in the methodology and methodologists in the logical and mathematical support of software technology[7].

We assert that the cause of the gaps and subsequent disappointment is not in inherent non-applicability of mathematical logic and algebra to software engineering but rather in using unsuitable mathematical tools: not the theory is unsuitable but an unsuitable theory is not suitable.

The following (oversimplified) picture of software evolution and mathematics employed traces the history of the situation (see Fig. 4).

Certainly, any attempt to describe the software world by a concise graphical image is doomed to be more or less speculative. Nevertheless, some global threads can be identified and are shown in the picture: the infamous impedance mismatch between databases (DB) and programming languages (PL) arisen in seventies, the trend of modern computational procedures to operate on extremely rich data structures and the trend of databases to be more flexible and rich from the computational view point. In fact, to manage modern ISs effectively one is forced to think in terms of *semantically valid computational procedures*: in effect, it joins databases and programming into an integrated object-oriented framework so that DB and PL tend to meet under the OO-supervision.

[7] as one of anonymous referees of our manifesto (Cadish and Diskin, 1995) wrote: "...none of the theorems (beyond the really basic stuff) is at all helpful"

As for the *Theory*-line, it appears that in computation modeling the powerful tools of category theory are used for attacking somewhat obsolete goal while in data modeling actual problems are attacked with unsuitable tools. Graph-based and object-oriented specifications are beyond the scope of the classical relational model but they are just in the focus of CT as we described briefly in section 3.1. Moreover, experience of CT shows that OO requires a special kind of thinking – *dynamic arrow thinking* in terms of (variable) sets and morphisms as opposed to static thinking in terms of sets and elements. Thus, the mathematics of (the formalizable part of) OO is CT and one has a nice amalgamation of modern trends in software with modern mathematics.

So, our general suggestion is to incorporate powerful CT-tools – the arrow logic and sketch language – into ISE very seriously from the very beginning, and thus build a unified specification framework for modern information technologies. Of course, we recognize well that the subject of engineering is much wider than it can be seen through mathematical patterns, and ISE is not an exception. However, in everyday work an engineer necessarily uses *working models* of the constructs he deals with and reasons of them on the base of some (implicit rather than explicit yet) *working logic*. So, what we actually propose is to make the *arrow thinking* underlying the sketch framework a working way of thinking in ISE. Being aware of the risk of making general statements and the more so predictions about the software world, we nevertheless assert that such a step should result in a significant increasing of information system design quality and productivity.

It seems for us that the current and of the nearest future situation in relating category theory with ISE theory and practice should be similar to the heroic age of the calculus in the 18th century when differentiation/integration techniques and mechanical engineering coupled closely in, in fact, a single discipline of theoretical mechanics and elasticity theory; only later preimages of modern purely mathematical disciplines had separated from that merge. The situation with CT and ISE is slightly different: the mathematics as such is already designated and developed, and we predict that in the nearest future it will be coupled with ISE. Initial steps have been done, and we conjecture that in some years the abstract "arrow nonsense" of category theory will become a basic mathematical discipline necessary for a software engineer very much like the ordinary linear algebra and calculus are for a mechanical engineer.

References

Atzeni, P. and Torlone, R. (1996). Management of multiple models in an extensible database design tool. In *Advances in Database Technology – EDBT'96*, 5th Int.Conf. on Extending Database Technology, Springer LNCS'1057, pages 79–95.

Batini, C., Battista, G., and Santucci, G. (1993). Structuring primitives for a dictionary of entity relationship data schemas. *IEEE Trans.Soft.Engineering*, 19(4):344–365.

Barr, M. and Wells, C. (1990). *Category Theory for Computing Science.* Prentice-Hall International Series in Computer Science. Prentice-Hall International, New York, 1990.

Cadish, B. and Diskin, Z. (1995). Algebraic Graph-Oriented = Category Theory Based. Manifesto of categorizing database theory. Technical Report 9506, Frame Inform Systems. //ftp.cs.chalmers.se/pub/users/diskin/MANIFEST/mnfst4.ps

Cadish, B. and Diskin, Z. (1996). Heterogeneous view integration via sketches and equations. In *Foundations of Intelligent Systems*, Proc. 9th Int.Symposium, *ISMIS'96*, Springer LNAI'1079, pages 603–612.

Diskin, Z. (1997a). Formalization of graphical schemas: General sketch-based logic vs. heuristic pictures. In *10th Int.Congress of Logic,Methodology and Philosophy of Science.* Kluwer Acad.Publ.

Diskin, Z. (1997b). Formalizing schemas for federal database environment architecture. Technical Report 9701, Frame Inform Systems, Riga, Latvia.
(On ftp: //ftp.cs.chalmers.se /pub/users/diskin/REPORTS/tr9701.ps).

Diskin, Z. (1997c). Generalized sketches as an algebraic graph-based framework for semantic modeling and database design. Technical Report 9703, Frame Inform Systems/LDBD. (//ftp.cs.chalmers.se/pub/users/diskin/REPORTS/tr9703.ps)

Diskin, Z. (1998a). The arrow logic of meta-specifications: a formalized graph-based framework for structuring schema repositories. In *Seventh OOPSLA Workshop on Behavioral Semantics of OO Business and System Specifications)*, Technical Report, Munich University. To appear.
(On ftp: //ftp.cs.chalmers.se/pub/users/diskin/PAPER-DB/ oopsla98.ps).

Diskin, Z. (1998b). The arrow logic of visual modeling and taming heterogeneity of semantic models. In *Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, Technical Report, Munich University.

Diskin, Z. and Kadish, B. (1995). Variable sets and functions framework for conceptual modeling: Integrating ER and OO via sketches with dynamic markers. In *OOER'95: Object-Oriented and Entity-Relationship Modeling*, Proc. 14th Int.Conf., Springer LNCS'1021, pages 226–237.

Diskin, Z. and Kadish, B. (1996). Variable set semantics for generalized sketches: Why ER is more object-oriented than OO. To appear in *Data and Knowledge Engineering*, manuscript is available by ftp //ftp.cs.chalmers.se/pub/users/diskin/ER/ERvsOO.ps.

Diskin, Z. and Kadish, B. (1997). A graphical yet formalized framework for specifying view systems. In *Advances in Databases and Information Systems, ADBIS'97*, 1st East-European Symposium.

Drew, P., King, R., McLeod, D., Rusinkiewicz, M., and Silberschatz, A. (1993). Report on the workshop on semantic heterogeneity and interoperation in multidatabase systems. *SIGMOD Record*, 22(3):47–56.

Francalanci, C. and Pernici, B. (1994). Abstraction levels for entity-relationship schemas. In *13th Int.Conf. ER'94*, Springer LNCS'881, pages 456–473.

Goguen, J. (1994). Requirements engineering as the reconciliation of social and technical issues. In *Requirements Engineering: Social and Technical Issues.* Academic.

Goguen, J. (1996). Formality and informality in requirement engineering. In *Requirement engineering, 4th Int. Conference*, pages 102–108. IEEE Computer Society. (keynote address).

Jifeng, H. and Hoare, C. (1989). Categorical semantics for programing languages. In *Mathematical foundations of programing semantics*, Springer LNCS'442, pages 402–417.

Johnson, M. and Dampney, C. (1993). On the value of commutative diagrams in information modeling. In *Algebraic Methodology and Software Technology, AMAST'93.* Springer.

Langmaack, H., Pnueli, A., de Roever, W.-P., and Strabner, A. (1997). Foreword. In *Compositionality. Proceedings of Int. Symposium.*

Makowsky, J. (1992). Model theory and computer science: An appetizer. In Abramsky, S., Gabbay, D., and Maibaum, T., editors, *Handbook of Logic in Computer Science*, volume 1. Oxford University Press.

Moggi, E. (1991). A category-theoretic account of program modules. *Mathematical structures in Computer Science*, 1:103–139.

OMG (1997). *UML Document Set.* Object management group, OMG, OMG's webpage, `http://www.omg.org/library/schedule/Technology-Adoptions.htm`.

Piessens, F. (1996). *Semantic data specifications: an analysis based on a categorical formalization.* PhD thesis, Dept. of Computer Science, Katholieke Universiteit Leuven.

Piessens, F. and Steegmans, E. (1997). Proving semantical equivalence of data specifications. *J. Pure and Applied Algebra*, (116):291–322.

Santucci, G., Batini, C., and Battista, G. (1993). Multilevel schema integration. In *12th Int.Conf. ER'93*, Springer LNCS'823, pages 327–338.

Wadler, P. (1992). Comprehending monads. *Mathematical structures in Computer Science*, 2(4).