

The Arrow Manifesto: Towards software engineering based on comprehensible yet rigorous graphic specifications

Zinovy Diskin¹, Boris Kadish¹,
Lab. for Database Design,
Frame Inform Systems, Ltd.
Riga, Latvia
www.fis.lv/english/science
diskin@fis.lv

Frank Piessens²
Dept. Of Computer Science
Katholieke Universiteit Leuven
Heverlee, Belgium
Frank.Piessens@cs.kuleuven.ac.be

1. Why: The problem of specifying

Computer is enigma. Rates and scales of its expansion into the everyday life and of the internal development are exponentially growing in multiple, diverse and hardly predictable directions. The most mysterious computer's component is software despite its precise logical foundations and origin. However, the logic of software is inherently *operational* and responds to the question *how* while *denotational* semantic meaning (*what*) is often implicit and remains hidden in the context. This results in a evident peculiarity of the situation: software expansion is accompanied with an extremely wide experimental activity but is based on surprisingly weak semantic specificational foundations (cf. [1]).

This peculiarity makes computer's intellect -- software -- somewhat similar to human's intellect whose productivity also often goes beyond strict denotational patterns. Human experience shows however that specificationless activity of the intellect can be productive only in the initial heuristic stages while advanced development needs clear specifying its goals. This is the more so for the logic-based software which seems has reached industrial maturity but is still in the state of youthful ignoring denotational specifications.

Certainly, the importance of specifications is well recognized and an abundance of specification languages was created. However, they do not meet the requirements of the software reality: we believe that weak specificational foundations of the software building are not caused by misunderstanding or underestimating the value of specifications as such but rather by the absence of suitable languages. Since any language, and even merely a notational system, is a more or less direct reflection of the corresponding underlying logic, software is actually suffered 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.

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)

2. What: An ideal specification paradigm

The diversity of specification problems in software 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 language pattern $\mathbf{L}(\mathbf{p})$, depending on a set \mathbf{p} of some logical constructs considered as parameters so that different specification languages would be instances of the \mathbf{L} -pattern defined by the choice of parameters: given a problem domain A , a specification language L_A suitable for A may be built as $\mathbf{L}(\mathbf{a})$ for some $\mathbf{p}=\mathbf{a}$ appropriate for A . Another domain B needs another language $L_B = \mathbf{L}(\mathbf{b})$ and so on but the *diversity* of domains A, B, \dots results in a *variety* of languages L_A, L_B, \dots since all these languages are treated in a uniform way as instances of the same pattern \mathbf{L} .

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

- **Semantic capabilities.** \mathbf{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. The semantics of the \mathbf{L} -languages should be *formalizable* to rule out ambiguities, and to allow for computer assisted verification.
- **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_D , developed for a domain D 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 and replace it with some equally expressive but entirely novel \mathbf{L} -language. A better solution would be to simulate L_D in \mathbf{L} , that is, to find parameters $\mathbf{p}=\mathbf{d}$ such that syntax and intended semantics of $\mathbf{L}(\mathbf{d})$ would be close to those of L_D . Thus, \mathbf{L} should have the possibility to simulate a wide range of *ad hoc* languages.
- **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: graphical evidence and modularizability.
 1. *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 graphic specification language such that graphs themselves should be precise specifications suitable for implementation.
 2. *Modularizability.* It is clear that a working description of large, and of moderate size too, system cannot be a flat, one-piece, specification. Rather, it will be an integrated collection of manageable specification modules: viewpoint specifications and their refinements of different degree

of elaborating details. Thus, the "ideal" paradigm we are discussing should provide a flexible modularization mechanism based on view and refinement constructs.

- **Abstraction flexibility.** This term means the capability to (meta) specify complex, consisting of multiple components yet integrated specifications in a way independent of the types of component specifications. In particular, the key notions of view to data and refinement of data should be defined in some abstract data-model-independent way.

Specification methods currently used in software engineering fail to integrate the requirements stated above in a single framework. Roughly speaking, they are either string-based, and hence heavily comprehensible in the case of industrial size systems, or graph-based but hardly have precise semantics and usually are of very limited expressive power. As a typical example, a modern database designer is forced to alter between *Scylla* of easy-to-use but scarcely formalized graphic languages like ER-diagrams or UML-schemas, and *Charybdis* of logically clear and perfect but inflexible relational data model. Of course, there are more or less successful compromises between expressiveness and comprehensibility, e.g., the well known SQL, but as any compromise, they apt only in restricted situations and do not solve the problem as such.

Fortunately, a methodology and specification machinery ideally matching the ideal properties we have formulated, have been developed in mathematical category theory where a special *arrow diagram logic*, and even a special *arrow thinking* underlying it, were invented for specifying complex mathematical structures.

3. How: Arrow logic and sketches for semantic specifications

3.1. Arrow thinking and category theory

Category theory (CT) is a modern branch of abstract algebra. It was invented in the late forties and since that time has achieved a great success in providing a uniform structural framework for different branches of mathematics, including metamathematics and logic, and stating foundations of mathematics as a whole as well. CT should be also of great interest for 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 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 (that is,

expressible in a formal set theory) can be described in the arrow language as well. Moreover, the arrow language is proven to be an extremely powerful conceptual means: if basic objects 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 objects of the universe. In other words, as it was formulated by a founder of categorical logic Bill Lawvere, *to objectify means to mappify*.

The arrow language is extremely flexible. In function of context, objects and arrows can be interpreted by: sets and functions (in *data modeling*), object classes and references (*OO analysis and design*), data types and procedures (*functional programming*), propositions and proofs (*logic/logic programing*), 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 of *sketch* specifications.

3.2. Graph + Diagram Predicates & Operations = Sketch

The sketch specificational vocabulary is strict and compact. Sketches are graphical constructs consisting of three kinds of items:

- (i) nodes, to be interpreted as objects (e.g. , sets),
- (ii) arrows, to be interpreted as morphisms (e.g. , functions),
- (iii) marked diagrams, i.e., labeled collections of nodes and arrows, to be interpreted as predicates (e.g. , declared for diagrams of sets and functions).

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

It is important that semantics of various diagram predicates can be defined in a uniform way. To wit, there is some set of basic operations such that any predicate's semantics can be defined in terms of these operations and equations; as a result, any signature of diagram predicates can be considered as an essentially algebraic theory over some common set of basic operations. This makes it possible (at least, in principle) to relate and compare sketches in different signatures in some unified way.

Surprisingly, but the brief sketch vocabulary of only three basic primitives is sufficient to cover all the required properties listed in section 2. Indeed, sketches are graphically evident specifications yet their semantics can be described in precise formal terms, not less precise than, say, semantics of first order logic theories. As arrow-based specifications, sketches possess a natural modularization mechanism, in addition, constructs of view and refinement can be described by arrows in an abstract way [3]. Also, semantics of sketches used for data modeling is inherently object-oriented and sketches do allow to provide for key semantic modeling constructs of *IsA*, *IsPartOf*, various aggregation and qualification relationships with precise semantic meaning [5,4]. Finally, an important observation we made while applying sketches is that in a concrete application domain sketches normally appear as a precise formal refinement of some existing notation rather than an external imposition upon the domain. In particular, sketches can be seen as a far reaching generalization of functional-data-model-schemas and ER-diagrams in conceptual

modeling (see also [6]), interaction diagrams in process modeling and schema grids in meta-specification modeling.

4. Conclusion

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.

Of course, relations between mathematics and engineering is a highly non-trivial issue, and for SE this is as true as for any other engineering activity. Fig.1 presents an oversimplified picture of software evolution and mathematics employed: our intention is to trace the history of their current very-far-from-being-in-match relations and to evaluate perspectives of their future interaction.

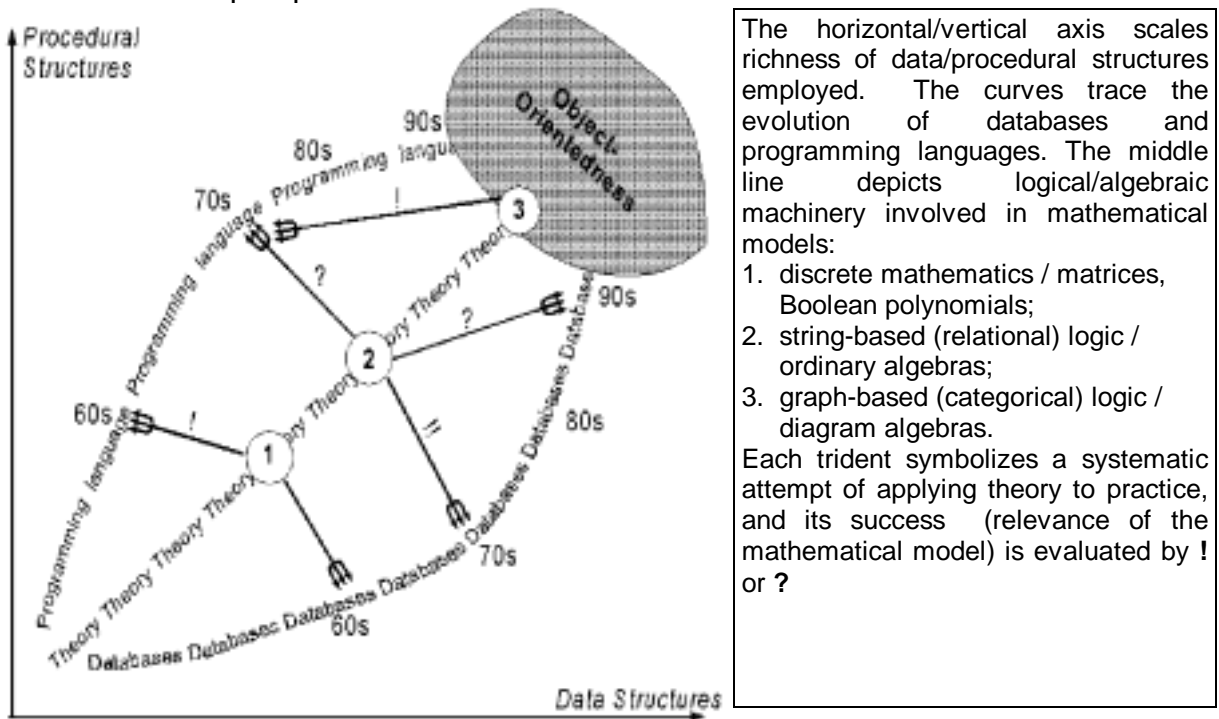


Figure 1: An overall picture of software evolution

Very briefly, the picture suggests that effective managing modern information systems leads to thinking in terms of *semantically valid computational procedures*: in effect, it joins databases (DB) and programming (PL) into an integrated object-

oriented framework. As for the *Theory*-line, it appears that currently in computation modeling the powerful tools of category theory are used for attacking somewhat obsolete goals while in data modeling actual problems are attacked with unsuitable tools (of mainly, relational data model). So, DB and PL tend to meet under the OO-supervision while mathematics of (the formalizable part of) OO is CT and thus, one has a nice amalgamation of modern trends in software with modern mathematics. Our general suggestion is to incorporate powerful CT-tools (the arrow logic and sketch language) into SE 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 SE 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 SE. 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 SE 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 SE 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 SE. 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

- [1] M. Atkinson, F. Bancilhon, D. DeWitt, K. Ditrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In: *The First Conference on Deductive and Object Oriented Databases*, Kyoto, Japan, 1989.
- [2] B. Cadish and Z. Diskin. Algebraic Graph-Oriented = Category Theory Based. Manifesto of categorizing database theory. Technical Report 9506, Frame Inform Systems, 1995 (On <ftp://ftp.cs.chalmers.se/pub/users/diskin/MANIFEST/mnfst4.ps>).
- [3] Z. Diskin. 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, 1998. To appear. (On <ftp://ftp.cs.chalmers.se/pub/users/diskin/PAPERS-DB/oopsla98.ps>)
- [4] Z. Diskin. The arrow logic of visual modeling and taming heterogeneity of semantic models. In *Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, Technical Report, Munich University, TU-19813, 1998. (On <ftp://ftp.cs.chalmers.se/pub/users/diskin/PAPERS-DB/ecoop98.ps>)
- [5] Z. Diskin and B. Kadish. Variable sets and functions framework for conceptual modeling: Integrating ER and OO via sketches with dynamic markers. In *OOER'95: Object-Oriented and Entity-Relationship Modeling*, Proc. 14th Int. Conf., Springer LNCS #1021, pages 226-237, 1995.
- [6] M. Johnson and C.N.G. Dampney. On the value of commutative diagrams in information modeling. In *AMAST'93: Algebraic Methodology And Software Technology*, Proc. 3rd Int. Conf., Springer LNCS, 1993