Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 203 (2008) 19–41
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Diagrammatic Logic for Object-Oriented Visual Modeling

Zinovy Diskin [1]

Department of Computer Sceince, University of Toronto, Canada
zdiskin@cs.toronto.edu

Uwe Wolter[2]

Department of Informatics, University of Bergen, Norway
wolter@ii.uib.no

Abstract

Formal generalized sketches is a graph-based specification format that borrows its main ideas from categorical and ordinary first-order logic, and adapts them to software engineering needs. In the engineering jargon, it is a modeling language design pattern that combines mathematical rigor and appealing graphical appearance. The paper presents a careful motivation and justification of the applicability of generalized sketches for formalizing practical modeling notations. We extend the sketch formalism by dependencies between predicate symbols and develop new semantic notions based on the Instances-as-typed-structures idea. We show that this new framework fits in the general patterns of the institution theory and is well amenable to algebraic manipulations.

Keywords: Diagrammatic modeling, model management, generic logic, categorical logic, diagram predicate, categorical sketch

1 Introduction

People like drawing pictures to explain something to others or to themselves. When they do it for software system design, they call these pictures *diagrams* or *diagrammatic models* and the very notation for them a *modeling language*. Syntax of diagrams is accurately specified in the so called metamodel but the intended meaning or semantics of diagrammatic constructs often remains intuitive and approximate. Sometimes it is so fuzzy that the construct becomes close to be meaningless at all,

[1] Supported by Bell Canada through the Bell University Labs, NSERC, and the Ontario Centres of Excellence.

[2] Research partially supported by the Norwegian NFR project SHIP/VERDIKT.

doi:10.1016/j.entcs.2008.10.041

in which case the experts advise to consider it as a modeling placebo (for example, this is how the construct of aggregation is treated in [23]). Until recently, this state of the art did not bother the modeling community too much: diagrammatic models were mainly used as a communication medium between business experts, software architects and programmers, and their precision was (although desirable) not a must.

The situation has dramatically changed a few years ago with the rapid invasion of the model-centric trends in software industry. Model-Driven Engineering (MDE), Model-Driven Development (MDD), Model-Driven Architecture (MDA) are different names of basically the same movement aimed at making models rather than code the primary artifacts of software development with code to be generated directly from models [24]. Needless to say that for MDD it is extremely important to have a precise formal semantics for diagrammatic notations. The industrial demand greatly energized building formal semantics for diagrammatic languages in use, and an overwhelming amount of them was proposed. A majority of them employ the familiar first-order (FO) logic patterns and string-based formulas, and result in bulky and somewhat unwieldy specifications. It is caused by the unfortunate mismatch between the string-based logical machineries and the internal logics of the domains to be formalized. Roughly, the latter are conceptually two-dimensional (graph-based) and class-oriented (are "sortwise") while the former are string-based and element-oriented (are "elementwise").

In the next section we discuss these problems in more detail and argue that the machinery of the so called *generalized sketches* proposed in [21,7,12], or the *Diagram Predicate logic* (DP-logic), offers just that apparatus which industry needs because it is inherently sortwise and graph-based (see also the discussion in [9]). We believe that as soon as modeling and modeling language design are becoming common tasks in software industry, DP-logic may become a *practical* logic for diagrammatic modeling in software engineering. That is why a clear presentation of DP-logic suitable for an engineer becomes an important task.

This paper has three main goals.

(a) The first is to motivate the applicability of the DP-logic pattern for formalizing practical diagrammatic notations used in software modeling. We show that generalized (rather than classical categorical) sketches appear on the scene quite naturally. In addition, it is very convenient to record some logical rules right in the signature of predicate symbols by introducing dependencies between the predicates [28].

(b) The second is to carefully define and explain this pattern in a way close to how a software engineer thinks of diagrammatic modeling. Particularly, it is important to "switch" from viewing semantics as a structure-preserving mapping *from* a specification to some predefined universe (the *indexed* view as it is customary in categorical logic) to the dual view of semantics as a structure-preserving mapping *to* a specification (the *fibrational* view). This is nothing but the idea of typing, which is ubiquitous in software engineering, and we will refer to this semantics as IATS (Instances As Typed Structures) (see [10] for the role of IATS in database metadata management). Mathematically, this switch is a special instance of the well-known duality between indexed and fibred categories.

As for the syntactic side of DP-logic, we tried to present it in a way parallel to how the syntactic basics of the ordinary FOL are usually presented. A corner-stone of this parallelism is a simple observation that a labeled diagram is nothing but a graph-based analog of a formula. More accurately, the notion of labeled diagram is quite generic and an ordinary logical formula $P(x_1...x_n)$ is just a specific syntax for a labeled diagram whose shape is the arity set $\alpha(P)$ of predicate P and the list $x_1...x_n$ encodes a mapping from $\alpha(P)$ to the set of variables used for building formulas. Then the notion of sketch naturally appears as a set of graph-based atomic formulas over a fixed set of names (variables). Table 1 below presents this and other parallels.

(c) Building DP-logic along the lines of (a,b) leads to a (quite natural yet) somewhat unusual logical formalism. It is not clear a priori whether it fits in the standard framework for "logic management" offered by the institution theory [17]. Thus, our third goal is to investigate whether DP-logic gives rise to an institution. In the IATS semantics, forgetful functors are defined by pullbacks and semantics becomes functorial only "up to isomorphisms". That is, in the fibred semantics setting, we cannot expect more than that the forgetful functors between categories of models constitute an overall lax (or pseudo) "model functor". We will indeed show that DP-logic provides a pseudo institution for any fixed signature of diagram predicates $\mathbf{\Pi}$. We will also show that $\mathbf{\Pi}$-sketches and their instance semantics form a lax specification frame [14].

In more detail, the contents of the paper is as follows. In section 2 we carefully motivate essential features of the machinery we are going to define: what are the benefits of classical categorical sketches, why we need their modification to generalized sketches, and why it is convenient to introduce dependencies between predicate symbols. In addition, we argue for IATS semantics as opposed to indexed semantics. In section 3 we first consider and discuss two simple examples of modeling with sketches, and then (subsection 3.2) discuss how to specify systems of models/sketches in the institution framework. These two sections aim mainly at goals (a) and (b). Section 4 presents a framework of accurate definitions and immediate results based on them, and culminates in Theorem 4.16 and 4.19 stating the main results described in (c) above.

2 A quest for logic convenient for diagrammatic modeling

2.1 Categorical sketches vs. first-order logic.

A key feature of universes modeled in software engineering is their fundamental *conceptual two-dimensionality* (further referred to as 2D): entities and relationships, objects and links, states and transitions, events and messages, agents and interactions; the row can be prolonged. Each of these conceptual arrangements is quite naturally represented by a graph – a 2D-structure of nodes and edges; the latter are usually directed and appear as arrows. In addition, these 2D-structures capture/model different aspects of the same whole system and hence are somehow interrelated between themselves. (For example, events happen to objects when the latter send and receive messages over dynamic links connecting them. These events trigger transitions over objects, which change their states). Thus, we come to another graph-based structure on the metalevel: nodes are graphs that model different aspects of the system and arrows are relations and interactions between them. The specificational system of aspects, views and refinements can be quite involved and results in a conceptually multi-dimensional structure. However complicated it may seem, this is the reality modern software engineers are dealing with (and languages like UML try to specify).

Describing this multidimensional universe in terms of FO or similar logics, which are based on string-based formulas talking about elements of the domains rather than their relationships, flattens the multi-level structure and hides the connections between the levels. This results in bulky and unwieldy specifications, which are difficult (if at all possible) to understand, validate, and use.

A radically different approach to specifying structures, which focuses on relationships between domains rather then their internal contents and is essentially graph-based, was found in category theory (CT). It was originated by Charles Ehresmann in the 60s, who invented the so called *sketches* (see [26] for a survey); later sketches were promoted for applications in computer science by Barr and Wells [2] and applied to data modeling problems by Johnson and Rosebrugh [19]. The essence of the classical sketch approach to specifying data is demonstrated by Fig. 1.

Figure 1(a1) shows a simple ER-diagram, whose meaning is clear from the names of its elements: we have a binary relation *O(wner)ship* over the sets *House* and *Person*, which also has an attribute *date*. In addition, the double frame of the node *House* denotes a so called *weak entity* [25]: there are no *House*-objects besides those participating in the relationship. A sample of another notation for data modeling, the now widely spread UML class diagrams, is shown in Fig. 1(a2). The edge between classes *House* and *Person* is called an *association*; labels 0..* and 1 near *association ends* are *multiplicity constraints*. They say that a *Person*(-object) can own any number of *House*-s including zero, and any *House* is owned by exactly one *Person*.

Evidently, data described by models can be seen as a configuration of sets and mappings between them. In the classical Ehresmann's sketch framework, this can be specified as shown in column (b) of Fig. 1. The upper sketch (b1) graphically consists of three pieces (above and below the dashed line) but actually consists of the carrier graph (the graph above the line) and a few labeled diagrams. The label "limit" is hung on the arrow span $(H \times P, p, o)$ (note the double arc) and declares the span to possess a special *limit* property. This property makes the set $H \times P$ the Cartesian product of *House* and *Person* (see, e.g., [2] for details). Similarly, the set *Date* is declared to be the Cartesian cube of the set *Integer* of natural numbers. Two additional diagrams below the dashed line force the arrows *in* and

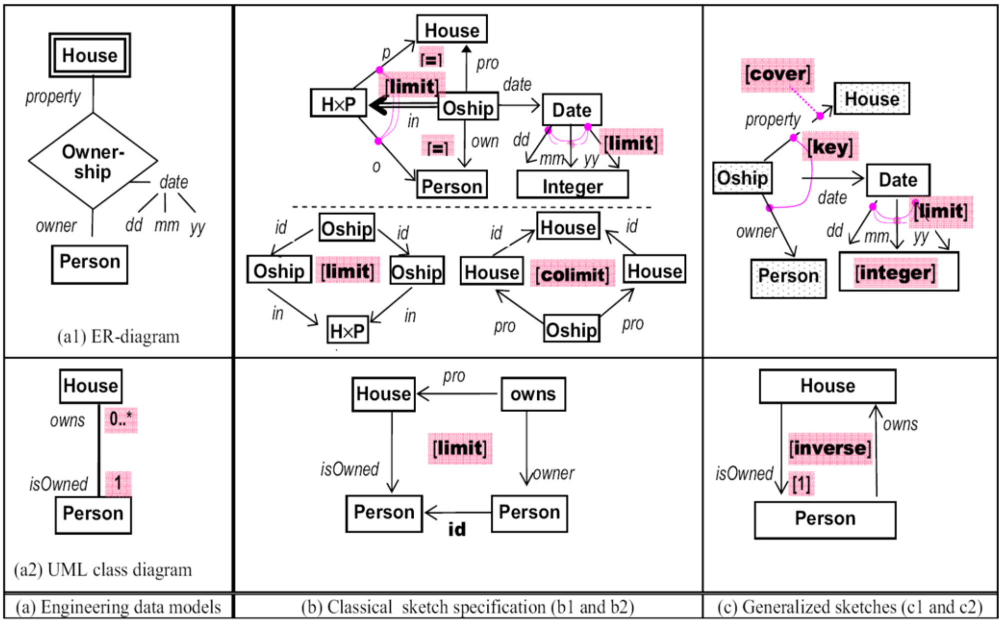


Fig. 1. A sample of sketching diagrammatic notations

pro to be, respectively, injective and surjective by standard categorical arguments [2]. Particularly, it implies that the set *Oship* is a subset of Cartesian product and hence is a relation. In other words, an *Oship*-object is uniquely determined by a *House*-object and a *Person*-object. Finally, two labels [=] declare the corresponding diagrams of mappings to be commutative so that mappings *pro* and *own* are indeed projections of the relation *Oship*. In the lower sketch (b2), the lower arrow is the identity mapping of set *Person* (the label "id" is a predicate rather than an arrow name), and hence the label "limit" declares the set *owns* with two projections to be the graph of mapping *isOwned*. This is basically the meaning of the UML diagram (a2): mapping *isOwned* is total and single-valued (the default property of arrows in column (b)) while mapping *owns* is a relation inverse to *isOwned*.

The limit and colimit predicates are amongst the family of the universal properties of sets-and-mappings diagrams. This family of predicates is extremely expressive and allows us to specify arbitrary properties of arbitrary sets-and-mappings configurations.[3] Particularly, if the configuration in question specifies the semantics of a diagrammatic model D (say, a ER or UML diagram), the corresponding sketch $\mathcal{S}(D)$ appears as a precise formal counterpart of D. We will call this procedure *sketching* the diagrammatic models. It provides a powerful mathematical framework for formalization and analysis of their semantics, see for example [19,22] for several useful results about ER-diagrams.

[3] In fact, since formal set theories can be encoded by universal predicates (by mapping them into toposes), we can say that any formalizable property of sets-and-mappings configurations can be expressed by universal diagram predicates and thus represented in the sketch language.

2.2 *Why* generalized *sketches?*

Although mathematically elegant, the classical sketch approach has several inherent drawbacks in engineering applications. For instance, in order to declare a simple fact that *O(wner)ship* is a binary relation, we were forced to introduce a few auxiliary elements into our specification. A similar complication occurred for specifying a simple surjectivity property. Note also that while extensions of nodes *House* and *Person* are to be stored in the database implementing the specification, extension of node $H \times P$ is (fortunately!) not stored. Similarly, classical sketch (b2) looks much more complicated than the original UML diagram. Thus, before we assign a precise semantic meaning to ER- or UML-diagrams, we need to apply to them some non-trivial transformations, and only after that the patterns of categorical logic can be used. From the view point of a software engineer, these transformations look artificial, unnecessary, and misleading.

Fortunately, the deficiency of the classical sketch framework mentioned above can be fixed while preserving the benefits that algebraic logic brings to the subject. The idea is demonstrated in column (c) of Fig. 1. Consider the upper specification (c1). We still want to specify the type of *O(wner)ship*-elements externally via mappings rather than internally (as is done in FOL), but we do not want to introduce the Cartesian product $H \times P$ into the specification. The crucial observation that allows us to accomplish the task is well-known: for a given span of mappings, e.g., $O = (\mathit{Oship}, \mathit{property}, \mathit{owner})$, its head *Oship* is isomorphic to a relation iff the legs (projection mappings) possess a special property of being *jointly injective* or *jointly monic*. The latter means that for any two different objects of type *Oship*, at least one of the leg mappings gives two different values (see [16] for details of an abstract formulation). This property of a family of mappings/attributes is well known in database theory under the name of *key*. Thus, we declare the span O to be a key and come to the specification shown in Fig. 1(c1).

Note also that label [Integer] is not the name of the node but a predicate label declaring the extension of the node to be the set of integers. Thus, the specification in Fig. 1(c1) presents a graph G, in which four diagrams are marked by predicate labels taken from a predefined signature. If Π denotes the signature, we will call such specifications *generalized* Π-*sketches* or just (Π-)*sketches*. The semantic meaning of such a sketch can be given by a graph morphism $[\![\ldots]\!] \colon G \to \mathsf{Set}$ into the graph Set of sets and mappings, which is compatible with the predicate labels in the sense outlined above. For example, the span $([\![\mathit{property}]\!], [\![\mathit{owner}]\!])$ is a key.

Similarly, specification (c2) presents another graph G with two labeled diagrams: arrow *isOwned* is labeled with its multiplicity constraint, and the pair of arrows is labeled with predicate [$\mathsf{inverse}$] declaring the corresponding mappings to be mutually inverse. The semantics of such a specification is given by a graph morphism $[\![\ldots]\!] \colon G \to \mathsf{Rel}$ into the graph Rel of sets and partial multi-valued mappings (binary relations), which is compatible with the predicate labels. That is, mapping $[\![\mathit{isOwned}]\!]$ is totally defined and single-valued (because of the multiplicity [$\mathsf{1}$]) and mapping $[\![\mathit{owns}]\!]$ is inverse to it. From now on, the term *sketch* will mean generalized sketch, that is, a graph-based object endowed with diagrams labeled by predicate symbols. Sometimes, when we want to remind that our sketches are formal objects rather than informal pictures, we will write *formal sketches.*

2.3 Semantics for generalized sketches, engineeringly

The passage from classical to generalized sketches does not finish our quest for specification machinery suitable for the modern software modeling. The point is that viewing semantics of a sketch as a mapping into Set or another semantic universe is not customary for a software engineer. In more detail, this view works well for the value-type part of the model, in our example, the part consisting of nodes Integer and Date and arrows between them. The primitive types (Integer in our case) have a special predefined semantics, and it has a fixed implementation in the computer system. The situation with the class part of the models (the *Oship*-span in our example) is different. This part can be considered as a UML class diagram G with a span of directed associations, see Fig. 2(b). UML defines a semantic instance of a class diagram to be a graph O of objects (nodes) and links (arrows) between them, which are typed/classified by class and association names respectively, see Fig. 2(c), where type labels are shown after colon (and in violet with a color display). Mathematically, labeling amounts to a graph mapping $\tau\colon O \to G$ (with G being the graph presenting the class diagram), which must satisfy diagram predicates attached to G. For example, the upper instance is not valid because two different *Oship*-objects have the same value (*property,owner*) (predicate [key] is violated), and object MarbleVilla is not in the range of mapping *property* (predicate [cov] is violated too). Note that by inverting the mapping τ we come to mapping $G \to \mathsf{Rel}$ rather than Set. Indeed, if $f\colon A \to B$ is an arrow in graph G, the set of arrows $\tau^{-1}(f)$ in O represents, in general, a relation between the sets of nodes $\tau^{-1}(A)$ and $\tau^{-1}(B)$ in O rather than a single-valued mapping between them. For example, for the lower instance in Fig. 2(c2), $\tau^{-1}(\mathit{property}) = \{p_{12}, p_{22}, p_{31}, p_{32}\}$ is a binary relation from $\tau^{-1}(\mathit{Oship}) = \{O1, O2, O3\}$ to $\tau^{-1}(\mathit{House}) = \{\mathit{HillHouse},\ \mathit{MarbleVilla}\}$.

Semantics as IATS (Instances As Typed Structures) is ubiquitous in software engineering. For a software engineer it is customary to think of an instance of a model G (a graph or another structure) as a structure O similar to G, which amounts to a structure preserving mapping $\tau\colon O \to G$. Note also an unfortunate mismatch with use of the term "model" in logic and engineering. In software engineering, a model normally refers to a syntactic concept while a corresponding semantic concept is called a model's instance. In mathematical logic, syntactic constructs are normally called specifications or theories while *models* are their semantic counterparts, "a model of a theory". In the paper we will use the terms *models* and *instance* in the engineering sense.

2.4 Dependencies (arrows) between predicate symbols

For the relational interpretation of arrows, being a single-valued or/and a totally defined relation become special properties to be explicitly declared rather than taken for granted like in the Set-valued semantics. Hence, we need to include the corresponding predicates into our signatures for everyday modeling. However, single-valued and total mappings still play a special role, and some diagram predicates like, e.g., [key], assume that all the participating arrows are such. In other words, if a span is declared to be a key, all its legs are automatically assumed satisfying the predicates of being single-valued and total. We say that there are *dependencies* [key] $\vdash$ [tot] and [key] $\vdash$ [s-val]. Of course, less trivial dependencies between predicates are also possible. It follows then that a signature is a graph, whose nodes are predicate symbols and edges are dependencies between them. A simple example in Fig. 2(a) demonstrates the idea.

The signature consists of three predicate symbols of arity shape "arrow" and one predicate symbol of arity "binary span". In addition, there are four arrows r_i, $i = sk1, sk2, tk1, tk2$ between the predicates, whose arities are mappings between the arity shapes. For example, the arrow $r_{sk1}\colon$ [s-val] $\to$ [key] denotes dependency [key] $\vdash_{r_{sk1}}$ [s-val] and its arity $\boldsymbol{\alpha}(r_{sk1})$ is the graph mapping sending the only arrow of [s-val]'s arity to the left leg of [key]'s arity span. It means that if a span of arrows is declared to be a [key], then its left leg must satisfy the predicate [s-val]. To ensure this for the right leg, we introduce another dependency [key] $\vdash_{r_{sk2}}$ [s-val] with arity mapping sending the arrow in [s-val]'s arity to the right leg of [key]'s arity span. The same situation is for the predicate of being totally defined relation.

Declaring a span of arrows in graph G as a [key], for example, means to define a graph mapping $d : \boldsymbol{\alpha}$[key] $\to G$ from the arity shape of [key] into G. The dependency [key] $\vdash$ [s-val] entails then that any declaration $d : \boldsymbol{\alpha}$[key] $\to G$ causes a corresponding declaration $\boldsymbol{\alpha}(r_{sk1}); d : \boldsymbol{\alpha}$[s-val] $\to G$ of an [s-val] arrow. Note that, due to pre-composition, the direction of arity mapping becomes opposite to the direction of dependency. We have chosen here to formalize dependencies by arrows between predicate symbols going in the direction of arity mappings. A convenient mnemonics for this is to use a special arrow-head for arrows between predicate symbols as shown in Fig. 2(a). Thus, a signature is a graph (category) $\boldsymbol{\Pi}$ of predicate symbols and dependencies between them, which are endowed with arities: a graph $\boldsymbol{\alpha}(P)$ for a node/predicate $P \in \boldsymbol{\Pi}$ and a graph mapping $\boldsymbol{\alpha}(r)\colon \boldsymbol{\alpha}(P) \to \boldsymbol{\alpha}(Q)$ for an arrow/dependency $r\colon P \longrightarrow\!\!\!\dashv Q$. In the next section we will see an example of how such a signature could work.

3 Modeling via sketches.

In this section we first consider two simple examples of modeling with sketches, each one using sketches over a different base category, Set and Graph, where the arities of predicate symbols live, and then outline their generalization (section 3.3). The reader may find it helpful to look at the upper part of Table 1 while reading the examples. In section 3.4, we discuss how to specify *systems* of models in the sketch framework.

Graph-based logic	**String-based logic**
Diagrams/Formulas and Substitutions	
Arity shape of a predicate P, $\alpha P \in \mathsf{Graph}$	Arity set, $\alpha P \in \mathsf{Set}$. Elements of this set are usually named by natural numbers
Carrier object/structure, graph $G \in \mathsf{Graph}$	Set of variables/context, $\Gamma \subset Var \in \mathsf{Set}$
Structural element (node or arrow), $e \in G$	Variable, $x \in \Gamma$
Mapping of carriers, $s\colon G \to G'$	Variable substitution, $s\colon \Gamma \to \Gamma'$
Labeled diagram, $G \Vdash P(d)$, with $d\colon \alpha P \to G$	Atomic formula in a fixed context, $\Gamma \Vdash P(\mathbf{x})$, where $\mathbf{x} = x_1...x_n$, i.e., $\mathbf{x}\colon \alpha P \to \Gamma$
Set of labeled diagrams over a carrier G, $Fm(\mathbf{\Pi},G)$	Set of atomic formulas in a fixed context, $Fm(\Pi,\Gamma)$
Dependencies and Derivations	
Predicate dependency, $r\colon Q \longrightarrow\!\!\dashv P$ with arity substitution $r^{\alpha}\colon \alpha Q \to \alpha P$	(Meta-)inference rule r : $\frac{P(a,b,c)}{Q(b,b)}$ with (metavariables) a, b, c ranging over Γ
Derivation/labeled sequent, $r\colon Q(r^{\alpha}; d) \longrightarrow\!\!\dashv P(d)$	Labeled sequent $P(x_7, x_3, x_4) \vdash^{r} Q(x_3, x_3)$
Graph of all labeled diagrams and all labeled sequents over the carrier G, $\boldsymbol{Fm}(\mathbf{\Pi},G)$	Graph of all atomic formulas and all labeled sequents in a fixed context Γ
Sketch = Structure + Constrains, $\mathcal{G} = (G, \boldsymbol{T})$ with $\boldsymbol{T}$ a subgraph of $\boldsymbol{Fm}(\mathbf{\Pi},G)$ closed w.r.t. derivations	Theory, $\Gamma \Vdash T$ or $\vdash^{\Gamma} T$, where $T \subset Fm(\Pi, \Gamma)$ and is closed w.r.t. all applications of the inference rules

Table 1
String-based logics as "graph-based" logic based on sets rather than graphs: Syntax

3.1 DP-logic over Set*: Painting objects (or business modeling for dummies :)*

A typical situation one encounters in modeling is categorization of objects into classes or types. We have a set of objects O, a set of types T (which are just names/symbols) and a typing mapping $\tau\colon O \to T$. For example, if T consists of four labels red, blue, black, and white, then a typing mapping would classify the set of objects into "red", "blue", "black", and "white" instances. In the modern jargon, the set {red, blue, black, white} is called the *model* and any mapping τ as above is its *instance*.

So far, any mapping $\tau\colon O \to \{\mathsf{red}, \mathsf{blue}, \mathsf{black}, \mathsf{white}\}$ is considered to be a legal instance of our model. Suppose however that by some *business rule*, the number of red objects must be always less than the number of blue objects, and the number of black objects is less than that of white ones. To specify this requirement, we first introduce into our specification language a binary predicate P, and then add to our model two predicate declaration or formulas $P(\mathsf{red}, \mathsf{blue})$ and $P(\mathsf{black}, \mathsf{white})$.

In more detail, the predicate symbol P has the arity set consisting of two elements (placeholders), for example, 1 and 2. A typing mapping $\omega\colon \Omega \to \{1, 2\}$ is considered to be compatible with P iff $|\omega^{-1}(1)| < |\omega^{-1}(2)|$. Formally, semantics of our predicate symbol P ***is*** the set $[\![\, P \,]\!]$ of ω's satisfying the requirement, i.e., we set $[\![P]\!] \overset{\text{def}}{=} \{\omega\colon \Omega \to \{1,2\} \mid |\omega^{-1}(1)| < |\omega^{-1}(2)|\}$. In predicate declaration $P(\mathsf{red}, \mathsf{blue})$, the bracketed part denotes the mapping $d\colon \{1,2\} \to T$, $T = \{\mathsf{red}, \mathsf{blue}, \mathsf{black}, \mathsf{white}\}$, with $d(1) = \mathsf{red}, d(2) = \mathsf{blue}$. Now, an instance $\tau\colon O \to T$ satisfies the declaration $P(d)$ if its part over the range of d is an element of $[\![P]\!]$.

The business logic of our colored objects may require other predicates. For example, we may need a ternary predicate $Q(1,2,3)$ such that a mapping $\omega\colon \Omega \to \{1,2,3\}$ satisfies Q iff $|\omega^{-1}(1)| + |\omega^{-1}(2)| \leq |\omega^{-1}(3)|$. If we now add to our model the declaration $Q(\mathsf{red}, \mathsf{blue}, \mathsf{black})$, any model's instance τ in which the total of red and blue objects is greater than the number of black objects will be invalid/illegal.

Our business logic may be even more complicated in that the rule $Q(1,2,3)$ can be enforced only if the conditions $|\omega^{-1}(1)| > 10$ and $|\omega^{-1}(2)| > 10$ hold. In more precise terms, we introduce a unary predicate U with semantics $|\omega^{-1}(1)| > 10$, and then define the arity of predicate Q to be the set {1,2,3} endowed with two declaration $U(1)$ and $U(2)$. Thus, having a declaration $Q(\mathsf{red}, \mathsf{blue}, \mathsf{black})$ in our model automatically means that declarations $U(\mathsf{red})$ and $U(\mathsf{blue})$ are also included.

The description above can be summarized as follows. We have a set of predicate symbols $\{U, P, Q\}$, each assigned with its arity set $\boldsymbol{\alpha}(..)$: $\boldsymbol{\alpha}(U) = \{1\}, \boldsymbol{\alpha}(P) = \{1,2\}, \boldsymbol{\alpha}(Q) = \{1,2,3\}$. In addition, we have two *predicate dependencies* or *rules* $r, r'\colon U \longrightarrow\!\!| \; Q$, whose arities are mappings $\boldsymbol{\alpha}(r), \boldsymbol{\alpha}(r')\colon \boldsymbol{\alpha}(U) \to \boldsymbol{\alpha}(Q)$ between arity sets: $r^{\alpha}(1) = 1$ and $r'^{\alpha}(1) = 2$, where we write r^{α} for $\boldsymbol{\alpha}(r)$. Dependencies serve as inference rules for formulas in the following way. Having a formula $Q(a,b,c)$ with variables a, b, c ranging over $T = \{\mathsf{red}, \mathsf{blue}, \mathsf{white}, \mathsf{black}\}$, we infer from it formulas $U(a)$ and $U(b)$ by applying dependencies r, r', in fact, their arity mappings r^{α}, r'^{α}. We will write $r\colon U(a) \longrightarrow\!\!| \; Q(a,b,c)$ and $r'\colon U(b) \longrightarrow\!\!| \; Q(a,b,c)$.

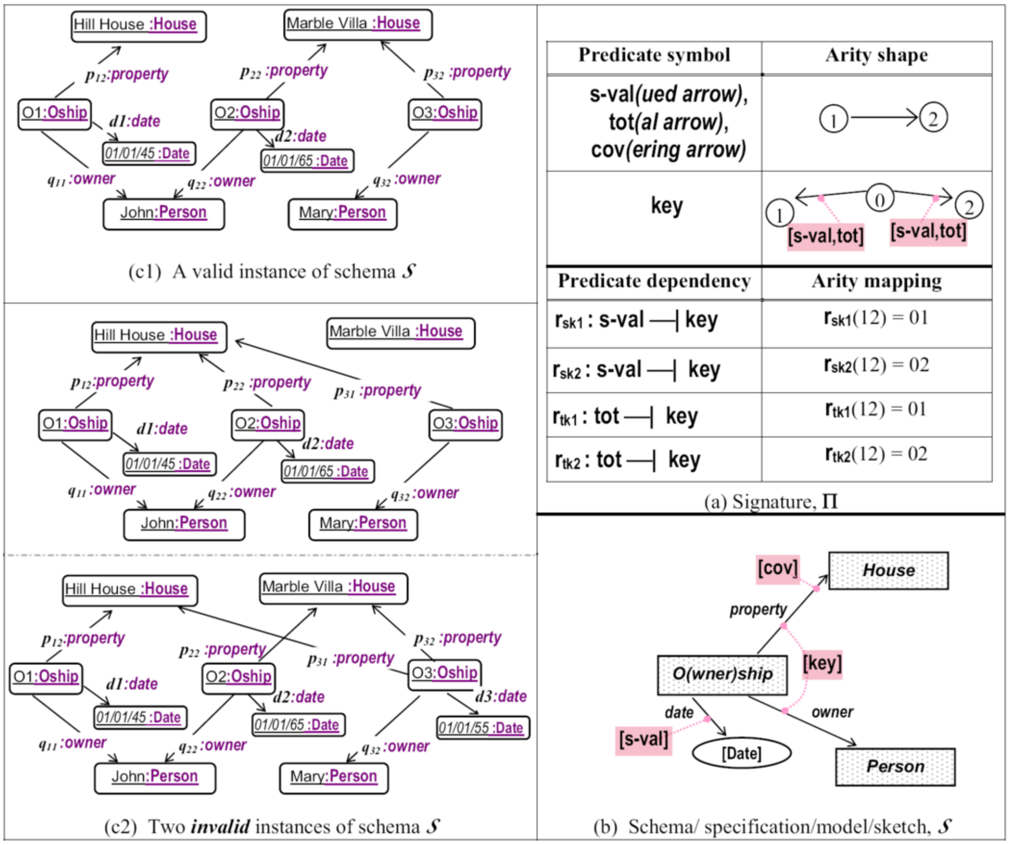
Thus, our logic for "painting objects" is based on the category Set of sets in the following sense. A signature is a graph (category) $\mathbf{\Pi}$ of predicate and dependency symbols, endowed with an *arity* graph mapping (functor) $\boldsymbol{\alpha}\colon \mathbf{\Pi} \to \mathsf{Set}$. Simultaneously, our models were *sets* of types, and instances are *sets* of objects together with typing as *set* mappings. Note that specifying simple cardinality constraints (like those considered above) in terms of classical sketches would be very bulky.

3.2 DP-logic over Graph*: Real estate via sketches*

We continue our discussion of the upper example in Fig. 1. The semantics of classes in the diagrams is specified by the formal sketch in the right column of Fig. 2: cell (a) presents the signature and cell (b) shows the sketch itself. The latter is a graph containing also three labeled diagrams:

$$[\mathsf{key}](\mathit{Oship}, \mathit{property}, \mathit{owner}),\ [\mathsf{cov}](\mathit{property}) \text{ and } [\mathsf{s\text{-}val}](\mathit{date}). \qquad (1)$$

We consider them as predicate declarations or formulas, whose round-bracketed parts actually encode the following mappings: $d_1\colon \boldsymbol{\alpha}[\mathsf{key}] \to G$ with $d_1(01) =$



Predicate symbol	Arity shape
s-val(*ued arrow*), tot(*al arrow*), cov(*ering arrow*)	$1 \rightarrow 2$
key	$1 \leftarrow 0 \rightarrow 2$ [s-val,tot] [s-val,tot]
Predicate dependency	**Arity mapping**
r_{sk1} : s-val —| key	$r_{sk1}(12) = 01$
r_{sk2} : s-val —| key	$r_{sk2}(12) = 02$
r_{tk1} : tot —| key	$r_{tk1}(12) = 01$
r_{tk2} : tot —| key	$r_{tk2}(12) = 02$

Fig. 2. Object instances as typed graphs

property, $d_1(02) = owner$[4]; $d_2 : \boldsymbol{\alpha}[\mathsf{cov}] \rightarrow G$ with $d_2(12) = property$ and $d_3 : \boldsymbol{\alpha}[\mathsf{s\text{-}val}] \rightarrow G$ with $d_3(12) = date$, where $\boldsymbol{\alpha}$ denotes the arity assignment and G denotes the carrier graph of the diagram (b) (that is, diagram (b) without labels). In addition, other four labeled diagrams are implicitly assumed in the sketch: they can be inferred by applying dependencies from the signature. For example, the sketch contains also the diagrams [s-val]($property$) and [tot]($property$).

An *instance* of the sketch is a graph O of objects and links typed by elements of the sketch so that the incidence between nodes and edges is preserved. A few samples of such typed graphs are presented in Fig. 2(c1,c2). Their elements are pairs $(e : t)$ with e denoting an element of O and t denoting its type (shown in violet with color display). In this way a typing mapping $\tau : O \rightarrow G$ is defined, and it is easy to verify that it is a graph morphism. In addition, this graph morphism must satisfy diagram predicates attached to G (and making it a sketch): $\tau \models P(d)$ for all diagrams $P(d)$ in the sketch.

Roughly, this means the following. The semantics of the label [cov] with arity

[4] Arrows in the arity graphs are named by the respective pairs of nodes, for example, 01 and 02. Note that the string-based notation used above in declaration (1) is ambigious and does not say whether $d_1(01) = property$ or $d_1(01) = owner$ but in this case it does not matter: the predicate [key] is symmetric.

$\boldsymbol{\alpha}[\mathsf{cov}] = \boxed{1 \longrightarrow 2}$ is a set of instances $\omega\colon \Omega \to \boxed{1 \to 2}$ such that $\omega^{-1}(12)\colon \omega^{-1}(1) \to \omega^{-1}(2)$ is a covering relation (surjection). The semantics of the label $[\mathsf{key}]$ is a set of instances $\omega\colon \Omega \to$ $$\begin{tikzcd} & 0 \arrow[dl] \arrow[dr] & \\ 1 & & 2 \end{tikzcd}$$ such that (i) relations $\omega^{-1}(01)$ and $\omega^{-1}(02)$ are totally defined and single-valued, and (ii) the pair of mappings $\langle \omega^{-1}(01), \omega^{-1}(02)\rangle$ is a key. With this semantics of predicate labels, Fig. 2 presents a valid (c1) and two invalid (c2) instances of the sketch in Fig. 2(b). [5]

Remark. We have used the inverse mapping τ^{-1} for the only sake to shorten the wording of elementwise specification. Our semantic definition is essentially elementwise rather than setwise. Nevertheless, after all our predicates speak about sets and mappings, and hence our logic is a logic of *sorts*, which is however implemented via elementwise specifications. An element-free specification is also possible via universal properties as it is done with Ehresmann's sketches, but a concrete implementation goes via elements.

3.3 Discussion: From models to formal sketches

We have considered examples of diagrammatic logical specifications over categories Set and Graph. Formalization of other diagrammatic notations used in software modeling leads to similar logical specifications based on these or similar graph-based structures. For example, a popular behavioral model, message sequence charts (MSCs) or close to them UML sequence diagrams, can be formally seen as mappings between graphs or 2-graphs (having arrows between arrows), see [11] for details. To manage this variety in a uniform way, we need a generic specification logic, whose syntactical apparatus is based on an arbitrary category **Base** from a wide class encompassing all interesting cases. For example, a good candidate for this class could be to consider **Base** to be an arbitrary presheaf topos. We call such a generic logic a *diagram predicate logic* over **Base**, and its specifications/theories are called (**Base**-) *sketches.*

A remarkable feature of the sketch in Fig. 1(c1) is its visual similarity to the original ER-diagram. We can even consider this diagram as nothing but a specific visual representation of the sketch, in which the diamond node in ER is just syntactic sugar for declaring the $[\mathsf{key}]$ predicate and in which the $[\mathsf{cover}]$ predicate for arrow *property* is visualized (in a somewhat misleading way) by double framing the target of the arrow. In the same way, the sketch in Fig. 1(c2) makes just explicit that the UML class diagram in (a2) declares actually two opposite multi-valued mappings representing the same binary relation.

In this way the generalized sketches treatment (*sketching* the diagrams) offers both (i) a precise formalization of their semantics and (ii) a framework that is visually appealing and transparent; as our experience shows, it can be readily understood by a software engineer. This is also true for the sketch formalization of

[5] The lower instance in (c2) is twice invalid: $\tau^{-1}(\mathit{property})$ is not surjective and the pair $\langle \tau^{-1}(\mathit{property}), \tau^{-1}(\mathit{owner})\rangle$ is not a key.

UML sequence diagrams [11], and to a lesser extent for sketching UML activity diagrams (a related discussion can be also found in [9]). These considerations and our practical experiments with sketching diagrammatic notations in [12,8,11] give rise to an ambitious thesis: a majority of diagrammatic notations really used in software engineering can be naturally seen as specific visualizations of the universal sketch specification pattern.

3.4 From systems of models to institutions built from sketches

Software development process normally results in a complex system of heterogenous models/specifications. Given a particular modeling language $\mathcal{L}$, e.g., that of ER-diagrams, or relational database schemas, or a sublanguage X of UML, we form the corresponding signature of diagram predicates $\mathbf{\Pi}_{\mathcal{L}}$ like it was shown above for ER-diagrams. Thus, we have signatures $\mathbf{\Pi}_{ER}$, $\mathbf{\Pi}_{Rel}$, $\mathbf{\Pi}_{UML[X]}$ and so on. Then $\mathcal{L}$-models can be formalized as $\mathbf{\Pi}_{\mathcal{L}}$-sketches, and their mappings as $\mathbf{\Pi}_{\mathcal{L}}$-sketch morphisms (defined below in Def. 4.6). Systems of similar models form so called *horizontal* sections of the entire model system. Various sorts of model translation, for example, generating logical relational schemas from ER-diagrams and physical schemas from logical ones, or Java code from high- through middle- and low-level UML-diagrams diagrams form the *vertical* dimension. In the sketch framework, this is formalized by mappings between sketches in different signatures. Clearly, design of these syntactical mappings and transformation should be based on their semantics. [6] Thus, we need to relate model mappings and transformations with models' instances and arrange it in a coherent mathematical framework. A standard pattern for such an arrangement is the notion of institution [17]. However, its application to the DP-logic, that is, relating institution's ingredients to DP-logic ingredients, is not straightforward.

In the context of the present informal discussion, it is convenient to call and denote the institution ingredients in the following way. What is usually called signatures, we will call *structure specifications*, or *i-signatures* to distinguish them from diagrammatic predicate signatures considered above. Correspondingly, i-sentences are called *constraints* and i-models are called *instances*. Thus, an institution is a quadruple $\mathcal{I} = (\mathbf{Str}, \boldsymbol{ctr}, \boldsymbol{inst}, \models)$ with $\mathbf{Str}$ a category of structure specifications, $\boldsymbol{ctr} : \mathbf{Str} \to \mathsf{Set}$ and $\boldsymbol{inst} : \mathbf{Str}^{op} \to \mathsf{Cat}$ are functors and $\models$ a family of binary satisfaction relations ($\models_S$: $S \in \mathbf{Str}$) satisfying the translation axiom

$$\boldsymbol{inst}(\sigma)(m_2) \models_{S_1} c_1 \quad \text{iff} \quad m_2 \models_{S_2} \boldsymbol{ctr}(\sigma)(c_1)$$

for all $\sigma : S_1 \to S_2$ in $\mathbf{Str}$ and all $c_1 \in \mathbf{Str}(S_1)$, $m_2 \in \boldsymbol{ctr}(S_2)$. A specification/theory/model is a pair $\mathcal{S} = (S, C)$ with $S \in \mathbf{Str}$ the structural part (an i-signature) and $C \subset \boldsymbol{ctr}(S)$ a set of constraints (i-sentences).

In applications we discussed above, the structural part of models is given by graphs, $\mathbf{Str} = \mathsf{Graph}$ or, in general, some category $\mathbf{Base}$ of graph-like objects (we

[6] Unfortunately, in the current state of the art of software development, models' semantics is implicit, which makes model management a very error-prone procedure [3]. It is one of our goals to build a sketch-based semantics for model management [13].

may think of **Base** as a presheaf topos). Constraints are diagrams labeled by predicate symbols from a predefined signature $\mathbf{\Pi}_{\mathcal{L}}$. Choosing a semantics interpretation for predicate symbols (see Def. 4.7 below) provides instances (Def. 4.12), which should complete this structure to an institution-like formalism (we will show it in the next section). Thus, an interpreted predicate signature determines an institution.[7] Correspondingly, sortwise signature morphisms are better to be considered together with their intended semantics interpretations and hence as institution morphisms. In this way, homogeneous (horizontal) model systems are arranged as institutions – each one generated by its own interpreted predicate signature $\mathbf{\Pi}$, while vertical model transformations are based on morphisms of $\mathbf{\Pi}$-signatures formalized as institutions morphisms.

In the present paper we deal only with horizontal model mappings and do not consider (vertical) signature morphisms. One reason is space limitations. Another reason is that it is appropriate to treat signature morphisms within a more advanced categorical presentation of the DP-logic than we are going to develop here (but will address in a forthcoming paper). Thus, in the next section we will build an institution for the DP-logic over fixed interpreted predicate signature $\mathbf{\Pi}$ and with **Base**-objects in the role of i-signatures, **Str**=**Base**. It can be done in a rather straightforward way similar to any other institution built for a functorial semantics logic, where the *reduction*, or *forgetful*, functors are simply defined by pre-composition. The only point that needs caution is that for our fibred semantics forgetful functors are defined by pullbacks rather than composition. Since pullbacks are determined only up to isomorphism, the i-model functor is lax, i.e., is a pseudo-functor.

4 Generalized sketches as a logical machinery

4.1 Syntax

This subsection presents the syntactic side of the DP-logic. The terminology is motivated mainly by the case of the base category being Graph, the category of directed (multi)graphs. However, to show parallels with ordinary string-based logics, we sometimes introduce two names for the same notion: graph-based and string-based logic motivated. Table 1 makes these parallels explicit. The reader is advised to consult with this Table while reading definitions below.

Let **Base** be some *base* category, arbitrary but fixed throughout the rest of the paper.

Definition 4.1 [Signatures] A *signature over* **Base** is given by a category $\mathbf{\Pi}$ of *predicate and dependency symbols* and a functor $\boldsymbol{\alpha}\colon \mathbf{\Pi} \to \mathbf{Base}$. For an object $P \in \mathbf{\Pi}$, the **Base**-object $\boldsymbol{\alpha}(P)$ is called the *arity (shape)* of P, and for a dependency

[7] We say that a signature is *interpreted* if it has some fixed predefined semantics in the sense of Def.4.7. Considering predicate signatures appearing in practice as interpreted is quite natural. For example, the predicate symbol [key] is like a predicate constant of arity "span" with predefined semantics of being jointly-monic.

arrow $r\colon Q \longrightarrow\!\!\!\!\dashv P$ in $\mathbf{\Pi}$, $\boldsymbol{\alpha}(r)\colon \boldsymbol{\alpha}(Q) \to \boldsymbol{\alpha}(P)$ is called an *arity substitution*. We will often write αP for $\boldsymbol{\alpha}(P)$ and r^α for $\boldsymbol{\alpha}(r)$.

We shall never deal with the situation of a few arity mappings defined on the same $\mathbf{\Pi}$, neither we will deal with signature morphisms in this paper. Hence, we can safely follow the terminological tradition of ordinary FOL and refer to a signature by the domain of the arity mapping.

Formulas in our generic logic are defined for a fixed *context* that is chosen independently of a given signature.

Definition 4.2 [Formulas and derivations] Let G be an object in **Base** to be thought of as the structural base/carrier graph of the specification.

(i) A *labeled diagram* or a *formula* over G is a pair $\varphi = (P, d)$ with P a predicate symbol and $d\colon \alpha P \to G$ a morphism (usually called a *diagram of shape* αP in the categorical jargon). Following notational traditions of string-based logics, we will denote formulas by expressions $P(d)$, and write $G \Vdash P(d)$ if we want to make the context/carrier object explicit. Let

$$Fm(\mathbf{\Pi}, G) = \{P(d) \mid P \in \mathbf{\Pi}, d \in \mathbf{Base}(\alpha P, G)\}$$

denote the set of all $\mathbf{\Pi}$-labeled diagrams/formulas over G.

(ii) A *derivation* over G is a triple $\langle r, Q(r^\alpha; d), P(d)\rangle$ with $r\colon Q \longrightarrow\!\!\!\!\dashv P$ a predicate dependency in $\mathbf{\Pi}$, and $P(d)$ a formula over G. We will write such a derivation as a labeled sequent, $r\colon Q(r^\alpha; d) \longrightarrow\!\!\!\!\dashv P(d)$ and say that it is produced by applying the dependency/rule $r\colon Q \longrightarrow\!\!\!\!\dashv P$ to $P(d)$. In this way, predicate dependencies serve as inference rules.

In general, the same formula $Q(r^\alpha; d)$ can be derived with another dependency $r'\colon Q \longrightarrow\!\!\!\!\dashv P$ if $r^\alpha; d = r'^\alpha; d$. Identical dependencies provide identical sequents since $(id_Q)^\alpha = id_{\alpha Q}$ for all predicate symbols Q. Moreover, we have $q^\alpha; p^\alpha = (q; p)^\alpha$ for all composable dependencies $q\colon Q \longrightarrow\!\!\!\!\dashv R$, $p\colon R \longrightarrow\!\!\!\!\dashv P$, and the associativity of composition in **Base** ensures that labeled sequents $q\colon Q(q^\alpha; (p^\alpha; d)) \longrightarrow\!\!\!\!\dashv R(p^\alpha; d)$ and $p\colon R(p^\alpha; d) \longrightarrow\!\!\!\!\dashv P(d)$ compose to a labeled sequent $q; p\colon Q((q; p)^\alpha; d)) \longrightarrow\!\!\!\!\dashv P(d)$. In such a way, the set of formulas together with the set of labeled sequents defines a category $\boldsymbol{Fm}(\mathbf{\Pi}, G)$ (note the bold font).

As most of specification formalisms, DP-logic offers translations of formulas caused by variable substitution. Let $s\colon G \to G'$ be a morphism in **Base**, which we may think of as a substitution of names/variables. The translation of formulas is based on the functor $\boldsymbol{s_*}\colon \mathbf{Base}{\downarrow}G \to \mathbf{Base}{\downarrow}G'$ defined by $\boldsymbol{s_*}(A, d) \overset{\text{def}}{=} (A, d; s)$ for all objects $(A, d : A \to G)$ in $\mathbf{Base}{\downarrow}G$ and by $\boldsymbol{s_*}(f) \overset{\text{def}}{=} f$ for all morphisms $f : (A, d) \to (B, e)$ in $\mathbf{Base}{\downarrow}G$.[8]

Construction 4.3 (Formula substitutions) *Given $s\colon G \to G'$ and formula $\varphi = P(d)$, we define $\boldsymbol{s_*}(\varphi) \overset{\text{def}}{=} P(\boldsymbol{s_*}(d)) = P(d; s)$. Substitution preserves derivations in*

[8] We remind that the slice category $\mathbf{Base}{\downarrow}G$ has pairs (A, d) with $d : A \to G$ as objects, and arrows $f : A \to B$ in **Base** such that $f; e = d$ as morphisms.

the following sense. Given a derivation

$$G \Vdash Q(r^\alpha; d) \xrightarrow{r} P(d),$$

we translate $P(d)$ *to* $\boldsymbol{s}_*(P(d)) = P(d;s)$ *and* $Q(r^\alpha; d)$ *to* $\boldsymbol{s}_*(Q(r^\alpha; d)) = Q[(r^\alpha; d); s]$*. By associativity of composition in* **Base***, the latter formula can be rewritten as* $Q[r^\alpha; (d; s)]$ *and hence we have a derivation*

$$G' \Vdash \boldsymbol{s}_*(Q(r^\alpha; d)) \xrightarrow{r} \boldsymbol{s}_*(P(d)).$$

It is easy to check that in this way a substitution $s\colon G \to G'$ gives rise to a functor $\boldsymbol{Fm}(s)\colon \boldsymbol{Fm}(\mathbf{\Pi}, G) \to \boldsymbol{Fm}(\mathbf{\Pi}, G')$ between formula categories.

Corollary 4.4 (Formula functor) *The assignments* $G \mapsto \boldsymbol{Fm}(\mathbf{\Pi}, G)$ *and* $s \mapsto \boldsymbol{Fm}(s)$ *define a* formula functor $\boldsymbol{Fm}_{\mathbf{\Pi}} : \mathbf{Base} \to \mathsf{Cat}$*.* [9]

We call specifications or theories in DP-logic *sketches*.

Definition 4.5 [Sketches] A *sketch over a signature* $\mathbf{\Pi}$ is a pair $\mathcal{G} = (G, T)$ with G the carrier graph (an object in **Base**) and $T \subset Fm(\mathbf{\Pi}, G)$ a set of diagrams/formulas over G. In addition, for any arrow $r\colon Q \longrightarrow P$ in $\mathbf{\Pi}$ the following inference condition must hold:

(Inf) *if formula* $P(d) \in T$ *then* $Q(r^\alpha; d) \in T$ *as well.*

In other words, the formula set T is closed under inference rules recorded in the signature and can be called a $\mathbf{\Pi}$-theory (hence, the letter T).

By taking the full category generated by T in the category $\boldsymbol{Fm}(\mathbf{\Pi}, G)$, we come to a *theory category* $\boldsymbol{T}$. Thus, a sketch can be considered as a pair $\mathcal{G} = (G, \boldsymbol{T})$ with $\boldsymbol{T}$ a subcategory of $\boldsymbol{Fm}(\mathbf{\Pi}, G)$ closed under all derivations (and hence being a full subcategory).

The usual requirement for a morphism between specifications that the target specification has to entail the translated source specification is reflected in DP-logic by the existence of functors.

Definition 4.6 [Sketch morphisms] A *morphism* $s\colon \mathcal{G} \to \mathcal{G}'$ between sketches $\mathcal{G} = (G, \boldsymbol{T})$ and $\mathcal{G}' = (G', \boldsymbol{T}')$ is a substitution $s\colon G \to G'$ such that the translation of formulas $\boldsymbol{Fm}(s)\colon \boldsymbol{Fm}(\mathbf{\Pi}, G) \to \boldsymbol{Fm}(\mathbf{\Pi}, G')$ restricts to a functor $\langle\!\langle \boldsymbol{s} \rangle\!\rangle\colon \boldsymbol{T} \to \boldsymbol{T}'$.

The category of sketches and their morphisms over a fixed signature $\mathbf{\Pi}$ will be denoted by $\mathbf{Ske}(\mathbf{\Pi})$.

4.2 Semantics of a sketch

Consider an object $G \in \mathbf{Base}$ as a model (in the engineering sense). Then, in the fibred (IATS) view of semantics, any morphism $\tau\colon O \to G$ is an *instance* of/over G,

[9] Note, that in contrast to most institutions presented in the literature, the formula functor is a functor into Cat and not only into Set.

$$\begin{tikzcd} O^* \arrow[r, "r^{\alpha *}"] \arrow[d, "\tau^*"'] \arrow[dr, phantom, "\text{[PB]}"] & O \arrow[d, "\tau"] \\ \alpha Q \arrow[r, "r^{\alpha}"'] & \alpha P \end{tikzcd}$$

(a) semantics of dependencies

$$\begin{tikzcd} O^{**} \arrow[r, "r^{\alpha *}"] \arrow[d, "\tau^{**}"'] \arrow[dr, phantom, "\text{[PB]}"] & O^* \arrow[r, "d^*"] \arrow[d, "\tau^*"] \arrow[dr, phantom, "\text{[PB]}"] & O \arrow[d, "\tau"] \\ \alpha Q \arrow[r, "r^{\alpha}"'] & \alpha P \arrow[r, "d"'] & G \end{tikzcd}$$

(b) satisfaction relation

$$\begin{tikzcd} O^{**} \arrow[r, "d'^*"] \arrow[d, "\tau^{**}"'] \arrow[dr, phantom, "\text{[PB]}"] & O^* \arrow[r, "s^*"] \arrow[d, "\tau^*"] \arrow[dr, phantom, "\text{[PB]}"] & O \arrow[d, "\tau"] \\ \alpha P \arrow[r, "d'"'] & G' \arrow[r, "s"'] & G \end{tikzcd}$$

(c) sketch morphisms

Fig. 3. Semantic notions

thus the category of all instances over G is given by the slice category $\mathbf{Base}{\downarrow}G$. We will denote isomorphisms in this category by $\cong$.

An object G is a purely structural model without any constraints. In the DP-logic, the latter are diagrams in G, and we need to define the notion of satisfiability of diagrams by instances. We do this below via the operation of pullback, and hence we assume that **Base** has pullbacks. In more detail, for every arrow cospan (x, y) in **Base**, we choose a fixed pullback span $(x^*, y^*) = PB(x, y)$ with the following notational agreement. If x^*, y^* are "parallel" to x, y respectively, we write $x^* \overset{\text{def}}{=} PB_y(x)$ and $y^* \overset{\text{def}}{=} PB_x(y)$. Then the familiar lemma that composition of pullbacks is again a pullback takes the following form: $PB_{x_1;x_2}(y) \overset{i}{\cong} PB_{x_1}[PB_{x_2}(y)]$ with $i = i(x_1, x_2, y)$ a canonical isomorphism satisfying the corresponding coherence conditions.

Clearly, before defining the next ingredient for an institution, namely, the satisfaction relation between G-instances and diagrams over G, we have to fix a semantic interpretation for the predicate symbols in our signature.

Definition 4.7 [Semantics of Signatures] Given a signature $\boldsymbol{\alpha}\colon \mathbf{\Pi} \to \mathbf{Base}$, its *semantic interpretation* is a mapping $[\![\,..\,]\!]$, which assigns to each predicate symbol P a set $[\![\,P\,]\!] \subset \{\tau \in \mathbf{Base} \mid cod\tau = \alpha P\}$ of *valid* instances, where $[\![\,P\,]\!]$ is assumed to be closed under isomorphisms: $\tau \in [\![\,P\,]\!]$ implies $i;\tau \in [\![\,P\,]\!]$ for any isomorphism $i : O' \to O$ in **Base**. The semantic interpretation is *consistent* if for any dependency $r\colon Q \longrightarrow\!\!\!\shortmid\; P$ in $\mathbf{\Pi}$ and any valid instance $\tau\colon O \to \alpha P$ of P, the induced instance $\tau^* \overset{\text{def}}{=} PB_{r^\alpha}(\tau)$ is a valid instance of Q (Fig. 3a). Below we will assume consistency by default (see Remark 4.11).

Example 4.8 Consider the signature specified in Fig. 2. For a graph mapping $\tau\colon O \to G$, we form another graph mapping $\tau^{-1}\colon G \to \mathsf{Rel}$ into the category of sets and binary relations. For the predicates $P = [\mathsf{s\text{-}val}], [\mathsf{tot}], [\mathsf{cov}]$, $\tau \in [\![\,P\,]\!]$ iff, respectively, $\tau^{-1}(12)$ is a single-valued, totally defined, covering relation, where 12 is the only arrow of the arity graph of these predicates. For the predicate $P = [\mathsf{key}]$, $\tau \in [\![\,P\,]\!]$ iff (a) the legs of the span $(\tau^{-1}(01), \tau^{-1}(02))$ are totally defined single-valued mappings and (b) the span is jointly monic (i.e., is a key to the set $\tau^{-1}(0)$). Condition (a) ensures consistency.

Now we define satisfiability of formulas in semantic instances.

Definition 4.9 [Satisfaction relation] Let G be an object in **Base** and $\tau\colon O \to G$ its instance. We say that this instance *satisfies* a labeled diagram/formula $P(d)$ over G and write $\tau \models_G P(d)$, iff $\tau^* \overset{\text{def}}{=} PB_d(\tau) \in [\![\, P \,]\!]$ (see Fig. 3b).

In other words, an instance of some model satisfies a labeled diagram/formula over this model if the part of this instance over the diagram is a valid instance of its label. The consistency assumption ensures soundness w.r.t. derivations.

Proposition 4.10 (Soundness) *If* $\tau \models_G P(d)$ *and* $r\colon Q(r^\alpha; d) \longrightarrow\!\!\!\!\dashv P(d)$ *is a derivation, then* $\tau \models_G Q(r^\alpha; d)$.

Proof. $\tau \models_G P(d)$ iff $\tau^* = PB_d(\tau) \in [\![\, P \,]\!]$ and this implies $\tau^{**} = PB_{r^\alpha}(\tau^*) \in [\![\, Q \,]\!]$ due to consistency (see Fig. 3b). Morphisms τ^{**} and $PB_{r^\alpha;d}(\tau)$ are isomorphic in **Base**$\downarrow G$ since the composition of two pullbacks is again a pullback. By assumption, $[\![\, Q \,]\!]$ is closed under isomorphisms, and thus we obtain $PB_{r^\alpha;d}(\tau) \in [\![\, Q \,]\!]$, i.e., $\tau \models_G Q(r^\alpha; d)$. □

Remark 4.11 (Consistency) *Following the tradition of functorial semantics, the original definitions of semantics for generalized sketches have been based on indexed concepts. The obvious idea is to transform a chosen semantic universe U given by a category like* Set*,* Par*,* Graph *or* Cat *into a "semantic $\mathbf{\Pi}$-sketch" $\mathcal{U} = (U, \boldsymbol{U})$ by defining for each predicate symbol P in $\mathbf{\Pi}$ its semantic meaning, that is, the set of "all valid diagrams" $d : \alpha P \to U$. A model (in the logical sense, i.e., i-model in terms of section 3.2) of a $\mathbf{\Pi}$-sketch $\mathcal{G}$ is then given by a sketch morphism from $\mathcal{G}$ into $\mathcal{U}$ (see [6,7,27,28]).*

Such an indexed semantics can be transformed into a fibred semantics, as presented here, if the underlying category allows for a variant of the so-called Grothendieck construction [18].[10] *The Grothendieck construction turns composition into pullbacks, and Proposition 4.10 makes apparent that the consistency condition in Def. 4.7 is just the "fibred counterpart" of the requirement that $\mathcal{U} = (U, \boldsymbol{U})$ has to be a $\mathbf{\Pi}$-sketch, i.e., that $\boldsymbol{U}$ has to be closed under inference rules recorded in the signature (compare Def. 4.5). In other words, any indexed semantics, where the underlying category allows for a variant of the Grothendieck construction, provides a consistent fibred semantics for predicate signature in the sense of Def. 4.7.*

Definition 4.12 [Instances of a sketch] Let $\mathcal{G} = (G, \boldsymbol{T})$ be a sketch. Its *instance* is an instance $\tau\colon O \to G$ of the structural base, such that all formulas of the sketch are satisfied: $\tau \models P(d)$ for all $P(d) \in \boldsymbol{T}$. It gives us the set of all instances $\mathit{Inst}(\mathcal{G})$ of sketch $\mathcal{G}$.

Let $\mathbf{Inst}(\mathcal{G})$ denote the full subcategory of category $\mathbf{Inst}(G) = \mathbf{Base}\downarrow G$ generated by the set $\mathit{Inst}(\mathcal{G})$ of all instances of sketch $\mathcal{G}$.

[10] In [27], we have analyzed the variant for the category Graph.

4.3 Semantics of a system of sketches: An institution arrangement

We have already built the syntactic part of the institution with Corollary 4.4 (take formulas/diagrams as i-sentences). As for semantics, we begin with the following old result [15, p.16].

Lemma 4.13 (Forgetful Functor) *Any morphism* $s\colon G' \to G$ *in* **Base** *induces a functor* $\boldsymbol{s^*} : \mathbf{Base}{\downarrow}G \to \mathbf{Base}{\downarrow}G'$ *right-adjoint to the functor* $\boldsymbol{s_*}\colon \mathbf{Base}{\downarrow}G' \to \mathbf{Base}{\downarrow}G$, $\boldsymbol{s_*} \dashv \boldsymbol{s^*}$, *where* $\boldsymbol{s_*}(\tau) \overset{\text{def}}{=} PB_s(\tau)$ *for any instance* $\tau\colon O \to G$ *of* G *(see Fig. 3(c)).*

Since pullbacks are only determined up to isomorphism, we cannot obtain that for any morphisms $s'\colon G'' \to G'$, $s\colon G' \to G$, the functors $\boldsymbol{(s';s)^*}$ and $\boldsymbol{s^*};\boldsymbol{s'^*}$ are equal. We can, however, prove that the functors $\boldsymbol{(s';s)^*}$ and $\boldsymbol{s^*};\boldsymbol{s'^*}$ are *naturally isomorphic.* Moreover, we can prove that the family of those natural isomorphisms satisfies the corresponding coherence conditions that give rise to

Proposition 4.14 (Instance functor) *The assignments* $G \mapsto \mathbf{Base}{\downarrow}G$ *and* $s \mapsto \boldsymbol{s^*}$ *define an* instance *pseudo functor* $\boldsymbol{Inst} : \mathbf{Base}^{op} \to \mathsf{Cat}$.

The last ingredient of an institution, the so-called *satisfaction condition*, is ensured for the fibred semantics by the fact that pullbacks are closed under composition and decomposition, respectively.

Corollary 4.15 (Satisfaction Condition) *For any morphism* $s\colon G' \to G$ *in* **Base**, *any instance* $\tau\colon O \to G$ *of the structural base* G, *and any formula* $P(d')$ *over the structural base* G' *we have*

$$\tau^* \models_{G'} P(d') \quad \textit{iff} \quad \tau \models_G P(d';s).$$

Proof. ($\Rightarrow$) $\tau^* \models P(d')$ means, due to Definition 4.9 and Lemma 4.13, that $\tau^{**} = PB_{d'}(\tau^*) = PB_{d'}(PB_s(\tau)) \in [\![P]\!]$ (see Fig. 3(c)). The composition of two pullbacks is again a pullback thus $PB_{d';s}(\tau)$ and τ^{**} are isomorphic. This means that $PB_{d';s}(\tau) \in [\![P]\!]$ since $[\![P]\!]$ is closed under isomorphisms, and thus $\tau \models_G P(d';s)$ due to Definition 4.9.
($\Leftarrow$) $\tau \models_G P(d';s)$ means that $PB_{d';s}(\tau) \in [\![P]\!]$. We know, however, that the pullback $PB_{d';s}(\tau)$ can be factored through the pullback $\tau^* = PB_s(\tau)$, meaning that we have an isomorphims between $PB_{d'}(\tau^*) = PB_{d'}(PB_s(\tau))$ and $PB_{d';s}(\tau)$. This entails $PB_{d'}(\tau^*) \in [\![P]\!]$, since $[\![P]\!]$ is closed under isomorphisms, and thus $\tau^* \models P(d')$. □

Summarizing the development so far, we can formulate

Theorem 4.16 ((Pseudo) Institution) *Let* **Base** *be a category with an operation of pullback, and* $\boldsymbol{\alpha}\colon \mathbf{\Pi} \to \mathbf{Base}$ *be a signature over* **Base**. *Then the formula functor* $\boldsymbol{Fm}_{\mathbf{\Pi}} : \mathbf{Base} \to \mathsf{Cat}$, *the instance pseudo functor* $\boldsymbol{Inst} : \mathbf{Base}^{op} \to \mathsf{Cat}$ *and the family* $(\models_G \mid G \in \mathbf{Base})$ *of satisfaction relations constitute a pseudo institution.*

One of the insights of the theory of institutions is that the satisfaction condition allows to extend i-model functors to "generalized i-model functors" for specifications. For the DP-logic, the standard argumentation instantiates as follows.

Corollary 4.17 *If $s\colon \mathcal{G}' \to \mathcal{G}$ is a morphism between sketches $\mathcal{G}' = (G', \boldsymbol{T}')$ and $\mathcal{G} = (G, \boldsymbol{T})$ and $\tau\colon O \to G$ is an instance of $\mathcal{G}$, then $\boldsymbol{s^*}(\tau\colon O \to G) = \tau^*\colon O^* \to G'$ is an instance of $\mathcal{G}'$ (see Fig. 3(c)).*

Proof. For any formula $P(d')$ in $\boldsymbol{T}'$ we have $\tau \models_{G'} P(d'; s) = \langle\!\langle \boldsymbol{s} \rangle\!\rangle(P(d'))$ since s is a sketch morphism and (O, τ) is an instance of $\mathcal{G}$. This, however, implies $\tau^* \models_{G'} P(d')$ due to (one direction of) the satisfaction condition. □

Since $\mathbf{Inst}(\mathcal{G}')$ and $\mathbf{Inst}(\mathcal{G})$ are full subcategories of $\mathbf{Base}{\downarrow}G'$ and $\mathbf{Base}{\downarrow}G$, respectively, Corollary 4.17 ensures that the forgetful functor restricts to instances.

Proposition 4.18 (Forgetful functor for sketches) *For any sketch morphism $s\colon \mathcal{G}' \to \mathcal{G}$, the forgetful functor $\boldsymbol{s^*}\colon \mathbf{Base}{\downarrow}G \to \mathbf{Base}{\downarrow}G'$ restricts to a functor $[\![\, s \,]\!]\colon \mathbf{Inst}(\mathcal{G}) \to \mathbf{Inst}(\mathcal{G}')$.*

Finally, we can summarize Propositions 4.18 and 4.14 with our second main result.

Theorem 4.19 (Generalized instance functor) *For any base category $\mathbf{Base}$ with pullbacks and any signature $\boldsymbol{\alpha}\colon \mathbf{\Pi} \to \mathbf{Base}$ over $\mathbf{Base}$, the assignments $\mathcal{G} \mapsto \mathbf{Inst}(\mathcal{G})$ and $(s\colon \mathcal{G}' \to \mathcal{G}) \mapsto [\![\, s \,]\!]$ define a pseudo functor $\boldsymbol{Inst}\colon \mathbf{Ske}(\mathbf{\Pi})^{op} \to \mathsf{Cat}$.*

5 Historical remarks, relation to other and future work

5.1 Historical remarks.

An early attempt to apply categorical ideas to data modeling can be found in [20]. A real application of categorical logic to data modeling was described in [5], with a major emphasis on commutative diagrams and less on the universal properties. In the same context of data modeling, the machinery of generalized sketches was developed and applied in a few industrial projects in Latvia in 1993-94 [4], and the corresponding logic presented at Logic Colloquium'95 [7]. Even earlier, Michael Makkai came to the need to generalize the notion of Ehresmann's sketches from his work on an abstract formulation of Completeness Theorems in logic. Makkai attributed the immediate impetus for him to work out these ideas to Charles Well's talk at the Montreal Category Theory Meeting in 1992. Well's own work went in a somewhat different direction [1] while Makkai's work resulted in the notion (and the very term) of generalized sketch. Makkai also developed a corresponding mathematical theory that was first presented in his several preprints circulated in 1993-94, and summarized and published later in [21].

Relations between generalized sketches as they are understood by Makkai and other generalization of the Ehresmann's concept are discussed in the introduction to [21]. The *"Latvian"* version of the definition of ordinary sketches coincides with

Makkai's definition (but emphasizes the parallelism between sketches and sets of atomic formulas in FOL). In addition, the Latvian version defines sketches over signatures with operation symbols [6], which we do not consider in this paper.

The original definitions of sketch semantics [7,6,21] are based on the "indexed view" in the sense that semantics of sketches is given by morphisms *from* specifications into a semantic universe. Semantic notions based on the "fibrational view", i.e., the Instances-as-typed-structures idea, are developed in the present paper for the first time. An ongoing research project at University of Bergen and Bergen University College aiming at practical applications of generalized sketches made apparent instantly the need for dependencies. The corresponding formal concepts and results have been presented first at LSFA'06 (see [28]).

5.2 Future work

We see the present paper as a common root of two lines (in fact, trees) of future work. One is to adopt the sketch formalism for formalizing modeling languages actively used in software engineering practice and specified in the corresponding industrial standards. Especially intriguing is to try to adapt the sketch patterns for behavior modeling. We also plan to explore how naturally can sketches be used for addressing real practical problems.

On the other hand, the formalism poses many interesting and non-trivial mathematical problems. (A) Makkai has introduced "multisketches" in [21] and shown that the category of multisketches is isomorphic to a certain presheaf topos in all cases, where the base category is a presheaf topos itself (for example, the category Graph). This result is important for application and implementation of sketches, and we are going to extend Makkai's constructions and results to signatures with dependencies. (B) A fully fledged sketch logic going beyond dependencies is still to be developed. (C) "Sketch operations" are the key to formalize model transformations and other aspects in Model Driven Development. A full exposition of these operations has to be given where especially the relation to Graph Transformations and to the concept of axioms in [21] will be of interest. (D) Finally, there are different ways to reformulate the definitions and results in this paper in a more categorical way. An investigation in this direction seems to be not only of theoretical interest.

Acknowledgement

We are grateful to the anonymous referee for a careful review of the paper and stimulating critique.

References

[1] A. Bagchi and C. Wells. Graph-based logic and sketches. In *10th Int.Congress of Logic,Methodology and Philosophy of Science, Florence, 1995*, Florence (Italy), 1997. Kluwer Acad.Publ.

[2] M. Barr and C. Wells. *Category Theory for Computing Science.* Prentice Hall International Series in Computer Science, 1995.

[3] P. Bernstein, A. Halevy, and R. Pottinger. A vision for management of complex models. *SIGMOD Record*, 29(4):55–63, 2000.

[4] B. Cadish and Z. Diskin. Algebraic graph-oriented approach to view integration. Part I: Specification framework and general strategy. Technical Report 9301, Frame Inform Systems/LDBD, Riga, Latvia, 1993. http://citeseer.ist.psu.edu/340019.html.

[5] C.N.G. Dampney, M. Johnson, and G.P. Monro. An illustrated mathematical foundation for ERA. In C.M. Rattray and R.G. Clarke, editors, *The Unified Computation Laboratory*. Oxford University Press, 1992.

[6] Z. Diskin. Databases as diagram algebras: Specifying queries and views via the graph-based logic of skethes. Technical Report 9602, Frame Inform Systems/LDBD, Riga, Latvia, 1996. http://citeseer.ist.psu.edu/116057.html.

[7] Z. Diskin. Towards algebraic graph-based model theory for computer science. *Bulletin of Symbolic Logic*, 3:144–145, 1997. Presented (by title) at *Logic Colloquium'95*.

[8] Z. Diskin. Visualization vs. specification in diagrammatic notations: A case study with the UML. In *Diagrams'2002: 2nd Int. Conf. on the Theory and Applications of Diagrams*, Springer LNAI#2317, pages 112–115, 2002.

[9] Z. Diskin. Mathematics of UML: Making the Odysseys of UML less dramatic. In Ken Baclawski and Haim Kilov, editors, *Practical foundations of business system specifications*, pages 145–178. Kluwer Academic Publishers, 2003.

[10] Z. Diskin. Mathematics of generic specifications for model management. In Rivero, Doorn, and Ferraggine, editors, *Encyclopedia of Database Technologies and Applications*, pages 351–366. Idea Group, 2005.

[11] Z. Diskin, J. Dingel, and H. Liang. Scenario integration via higher-order graphs. Technical Report 2006-517, School of Computing, Queen's University, Kingston, ON, Canada, 2006. http://www.cs.queensu.ca/TechReports/reports2006.html.

[12] Z. Diskin and B. Kadish. Variable set semantics for keyed generalized sketches: Formal semantics for object identity and abstract syntax for conceptual modeling. *Data & Knowledge Engineering*, 47:1–59, 2003.

[13] Z. Diskin and B. Kadish. Generic model management. In Rivero, Doorn, and Ferraggine, editors, *Encyclopedia of Database Technologies and Applications*, pages 258–265. Idea Group, 2005.

[14] H. Ehrig and M. Große-Rhode. Functorial Theory of Parameterized Specifications in a General Specification Framework. *Theoretical Computer Science*, (135):221 – 266, 1994.

[15] P. Freyd. Aspects of topoi. *Bull. Austr. Math. Soc.*, (7):1–72, 1972.

[16] P. Freyd and A. Scedrov. *Categories, Allegories.* Elsevier Sciece Publishers, 1990.

[17] J. A. Goguen and R. M. Burstall. Institutions: Abstract Model Theory for Specification and Programming. *Journals of the ACM*, 39(1):95–146, January 1992.

[18] B. Jacobs. *Categorical logic and type theory.* Elsevier Science Publishers, 1999.

[19] M. Johnson, R. Rosebrugh, and R. Wood. Entity-relationship-attribute designs and sketches. *Theory and Applications of Categories*, 10(3):94–112, 2002.

[20] S.K. Lellahi and N. Spyratos. Towards a categorical data model supporting structured objects and inheritance. In *Next Generation Information Systems*, volume 504 of *Springer Lect.Notes in Comp.Sci*, Kiev, 1990.

[21] M. Makkai. Generalized sketches as a framework for completeness theorems. *Journal of Pure and Applied Algebra*, 115:49–79, 179–212, 214–274, 1997.

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

[23] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual. Second Edition.* Addison-Wesley, 2004.

[24] Bran Selic. Model-driven development: Its essence and opportunities. In *ISORC*, pages 313–319, 2006.

[25] D.C. Tsichritzis and F.H. Lochovsky. *Data models.* Prentice-Hall, Englewood Cliffs, N.Y., 1982.

[26] C. Wells. Sketches: Outline with references. Available under http://www.cwru.edu/artsci/math/wells/pub/papers.html.

[27] U. Wolter and Z. Diskin. From Indexed to Fibred Semantics – The Generalized Sketch File. Technical Report Report No 361, Department of Informatics, University of Bergen, October 2007.

[28] U. Wolter and Z. Diskin. The Next Hundred Diagrammatic Specification Techniques – An Introduction to Generalized Sketches. Technical Report Report No 358, Department of Informatics, University of Bergen, July 2007.