A Formal Approach to Modeling and Model Transformations in Software Engineering

Adrian Rutle[1], Uwe Wolter[2], and Yngve Lamo[1]

[1] Bergen University College, p.b. 7030, 5020 Bergen, Norway {aru,yla}@hib.no
[2] University of Bergen, p.b. 7803, 5020 Bergen, Norway Uwe.Wolter@ii.uib.no

Abstract. A software model is an abstract representation of a software system which can be used to describe, at a higher abstraction level, different aspects of the software system. Since the beginning of computer science, raising the abstraction level of software systems has been a continuous goal for many computer scientists. This has lead to the usage of models and modeling languages in software development processes. Currently in addition to documentation purposes, models are increasingly used to automatically generate and integrate parts of the systems that they describe. As a consequence, there is a need for formal modeling languages and formal transformation definition techniques which can be employed to automatically translate between (and integrate) models. Therefore, a major focus of our research is on the formalization of modeling and model transformation in the generic formalism, Diagrammatic Predicate Logic (DPL). This paper provides an overview of the state-of-the-art of our ongoing research on analysis of modeling and model transformations based on the DPL framework.

1 Introduction and Motivation

Models, model transformations and automatization of model transformations are key issues in the emergent approach of software development process, which is standardized by the Object Management Group (OMG) as Model Driven Architecture (MDA) [5]. In the MDA approach, building an application starts with a (set of) formal, platform-independent models (PIM) in which the structure, logic and behavior of the application are specified. The PIMs are then transformed by transformation tools to a set of platform-specific models (PSM). These PSMs are used as input to code-generation tools which automatically create software systems based on the input models [4].

In MDA, the application platform and the implementation technology are chosen independently of the input models. This provides the flexibility to migrate to new technologies without doing changes in the domain model. In addition, the domain model can be modified, in response to business changes, independently of the application platform.

The transformation processes in MDA are specified by transformation definition languages and are executed by transformation tools. In order to enable

different tools to understand the same transformations in the same way, the models and the transformations between them are required to be defined formally [4]. This implies the necessity of techniques which can be used to specify formal models and formal transformation definitions. In addition, using formal modeling techniques opens for the development of mechanisms for reasoning about models and model transformations; mechanisms for model de-composition and integration; as well as mechanisms for verification of correctness, consistency and validity of models and transformation definitions.

There are many modeling languages that are used to write formal domain models. However, most of these modeling languages are either not sufficiently formalized or very complicated (text-based) or both, which makes writing formal models difficult and error-prone [3]. Moreover, since software models are graph-based, modeling languages which use string-based logic instead of graph-based logic may fail to express all properties of software systems in an intuitive way [2]. Thus, diagrammatic modeling is a better approach for modeling software systems since it is graph-based – making the relation between the syntax and semantics of models more compact –, and nonetheless, it is easier for domain experts to understand.

However, diagrammatic modeling languages are considered more difficult to formalize than text-based languages, therefore, diagrammatic languages often use text-based languages to define constraints and system properties that are difficult to express by their own syntax and semantics, e.g. the combination of UML and OCL. This turns models to a mixture of text and diagrams which is often difficult for non-experts to evaluate and understand, i.e. the models loose their simplicity and high level of abstraction which are the most appealing features of modeling.

The challenges in our research are to find a mathematical foundation for terms and processes that are frequently used in conjunction with MDA. We have used the DPL framework to answer questions such as; What is a model? What is an instance of a model? How to check conformance of a model to its metamodel? What is a model transformation? What is a correct model transformation? How to categorize model transformations? How the effects of the execution of a model transformation are analyzed? Some of these questions are discussed in more details in this paper (Sectoin 2); and Section 3 concludes the paper.

2 The Approach for the Formalization

Our approach to the formalization is based on the generic formalism, Diagrammatic Predicate Logic (DPL)[3]. We considered DPL as a suitable specification formalism to define diagrammatic modeling languages with a strong mathematical foundation since it is a fully diagrammatic specification formalism which we believe is necessary for formalization of modeling and model transformations in

[3] The DPL framework is called Generalized Sketches in previous publications [8,7], however, since the concept of "sketch" is misleading in SE, the name of the formalism is changed to DPL.

Software Engineering (SE). This section summarizes the DPL framework and its usage in modeling and model transformations.

2.1 Modeling and DPL

DPL is a graph-based specification format that borrows its main ideas from both categorical and first-order logic, and adapts them to SE needs [8]. The DPL formalism is a generalization and adaptation of the categorical sketch formalism where signatures are restricted to a limited set of predicates: limit, colimit and commutative diagrams [6]. This generalization is necessary to make DPL suitable for use in SE.

Signatures, diagram specifications and instance of diagram specifications are the concepts in the DPL framework which we use to represent modeling languages, models and instances, respectively. The definition of these concepts are given as follows:

Definition 1. *A diagrammatic predicate signature* $\Sigma := (\Pi, ar)$ *is an abstract structure consisting of a collection of predicate symbols* Π *with a mapping that assigns an arity (graph)* $ar(p)$ *to each predicate symbol* $p \in \Pi$*, i.e.* $ar : \Pi \rightarrow Graph$*.*

Definition 2. *A diagram* (p, δ) *labeled with the predicate* p *in a graph* $G(S)$ *is a graph homomorphism* $\delta : ar(p) \rightarrow G(S)$*, where* $ar(p)$ *is the arity of* p*.*

Definition 3. *A* Σ*-specification* $S := (G(S), S(\Pi))$*, is a graph* $G(S)$ *with a set* $S(\Pi)$ *of diagrams in G(S) labeled with predicates from the signature* Σ*.*

Definition 4. *An instance of a diagram specification* S *is a graph* I *together with a graph morphism* $\iota : I \rightarrow G(S)$*, where* $G(S)$ *is the carrier graph of* S*, such that* ι^* *is a valid instance of* p *(based on a given semantics of* p*) for each diagram* $\delta : ar(p) \rightarrow G(S)$ *where* ι^* *is given by the pullback diagram in Figure 1.*

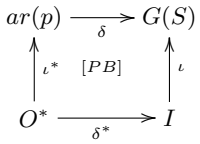


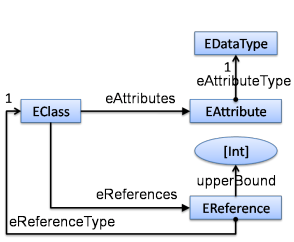
Fig. 1: Instance of the diagram specification S

In DPL, each (modeling) language L is represented by a formalism $F_L = (\Sigma_L, M_L)$ which consists of a diagrammatic signature Σ_L and a metamodel M_L. The signature Σ_L contains the language constructs and constraints that can be set by the language L, while the metamodel M_L (which is a diagram specification) specifies the syntactical structure of models that are allowed to be specified by the language. Moreover, software models that are specified by L are represented by Σ_L-specifications which conform to M_L [6].

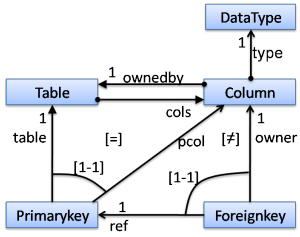
The diagram specification which represents the metamodel of a language L may, in cases where L is reflective, be a Σ_L-specification. However in general, a common formalism $F_C = (\Gamma, M)$ is used to specify the metamodels of the languages in the DPL framework.

Metamodeling is a mechanism for defining graphical modeling languages which is used in the way grammars in Bakus Naur Form (BNF) are used to define text-based languages such as programming languages [4]. A BNF grammar describes which series of tokens are valid expressions in a language. In the same way, a metamodel describes which graphs are valid models in a given modeling language. A fundamental difference here is that the representation of the structure of text-based languages is based on terms (abstract syntax trees,) while graphical languages have a graph-like structure which makes it impossible to apply BNF for their representation [1]. The metamodels of the graphical languages are usually represented by typed graphs.

Figure 2 shows an example of two diagram specifications which are representing a simplified metamodel of EMF –Ecore– (Figure 2a) and a simplified metamodel of Relational Databases (RDB) (Figure 2b). The details of some of the diagram predicates (and their semantics) from the signature Γ which are used to label parts of the diagram specifications in the figures are shown in in Table 1. For example primary key in the metamodel of RDB is a special property which every table in relational databases *must* have. This constraint is enforced by the predicate label [cover] on the arrow $table : PrimaryKey \to Table$, which is visualized as a filled arrow head. The set of columns which are specified as primary key must be unique for each table, i.e. each primary key belongs to only one table and each table has only one primary key. This constraint is enforced by the predicate labels $[1-1]$ and $[1]$ on the diagrams $Table \leftarrow PrimaryKey \to Column$, and $table : PrimaryKey \to Table$, respectively, meaning that each primary key is a relationship between a table and a set of columns which is uniquely identified by the set of columns and the table.



(a) A simplified diagram specification M_{EMF} of Ecore.



(b) A simplified diagram specification M_{RDB} of the metamodel of RDB.

Fig. 2: The simplified metamodels of EMF and RDB.

name	arity	visualization	semantic
[objectNode]	1	$\boxed{A}$	set of objects
[valueNode]	1	(A)	set of values
[total]	$1 \xrightarrow{x} 2$	$\boxed{A} \bullet\!\!\xrightarrow{f} \boxed{B}$	$\forall a \in A : \exists b \in B \mid b \in f(a)$
[key]	$1 \xrightarrow{x} 2$	$\boxed{A} \xrightarrow{f\ [key]} \boxed{B}$	$\forall a, a' \in A : a \neq a'$ implies $f(a) \neq f(a')$
[singlevalued]	$1 \xrightarrow{x} 2$	$\boxed{A} \xrightarrow{f\ \ 1} \boxed{B}$	$\forall a \in A : \lvert f(a)\rvert \leq 1$
[cover]	$1 \xrightarrow{x} 2$	$\boxed{A} \xrightarrow{f} \triangleright\boxed{B}$	$\forall b \in \wp(B) : \exists a \in A \mid b \in f(a)$
[inverse]	$1 \underset{y}{\overset{x}{\rightleftarrows}} 2$	$\boxed{A} \underset{g}{\overset{f}{\underset{[inv]}{\rightleftarrows}}} \boxed{B}$	$\forall a \in A$, $\forall b \in B : b \in f(a)$ iff $a \in g(b)$
[disjoint-cover]	$1 \xrightarrow{x} 2$, $2 \xrightarrow{y} 3$	$\boxed{A} \xrightarrow{f} \boxed{B} \xleftarrow{g} \boxed{C}$	$\bigcup\{f(a) \mid a \in A\} \cap \bigcup\{g(c) \mid c \in C\} = \emptyset$ and $\bigcup\{f(a) \mid a \in A\} \cup \bigcup\{g(c) \mid c \in C\} = B$
[jointly-mono] or [1-1]	$1 \xrightarrow{x} 2$, $1 \xrightarrow{y} 3$	$\boxed{A} \xrightarrow{f} \boxed{B}$, $\boxed{A} \xrightarrow{g} \boxed{C}$	$\forall a, a' \in A : a \neq a'$ *implies* $f(a) \neq f(a')$ or $g(a) \neq g(a')$
[=]	$1 \xrightarrow{x} 2$, $1 \xrightarrow{z} 3$, $2 \xrightarrow{y} 3$	$\boxed{A} \xrightarrow{f} \boxed{B}$, $\boxed{A} \xrightarrow{h} \boxed{C}$, $\boxed{B} \xrightarrow{g} \boxed{C}$	$h(a) = \bigcup\{g(b) \mid b \in f(a)\}$

Table 1: A signature Γ.

We say that a diagram specification I conforms to, or is typed by, its metamodel M if there is a graph homomorphism $t : G(I) \to G(M)$, where $G(I)$ and $G(M)$ are the carrier graphs of I and M respectively. While $\iota : I \to G(M)$ is said to be a valid instance of M iff for each diagram (p, δ) in $M(\Pi)$, the part of I related to the diagram δ, is a valid instance of p, for a given semantics of p. Examples of valid instances of a diagram predicate and a diagram specification are given in [6]. Thus, we distinguish between *conformance to* and *being instance of* (meta)models.

An example of an instance of the metamodel in Figure 2a is I_{EMF} which is shown in Figure 3. The nodes in this instance are typed by the nodes in M_{EMF}, e.g. `Person:EClass` and `worksFor:EReference`[4]. In addition, the diagram predicates in M_{EMF} are respected by I_{EMF}, e.g. the diagram $([total], \delta_1) : (\ 1 \xrightarrow{x} 2\ ) \mapsto (\ EReference \xrightarrow{upperbound} Int\ )$ is fulfilled in I_{EMF} which means that every node of type $EReference$ *must* have an upperbound (multiplicity) of type Int.

[4] Notice that `Person:EClass` is a "user-friendly" notation for the assignment ($\iota : Person \mapsto EClass$)

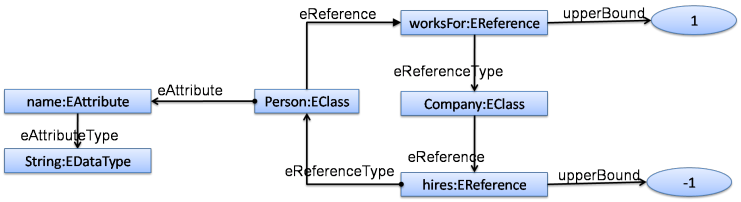


Fig. 3: An EMF model, I_{EMF}, which is an instance of the diagram specification M_{EMF} (Figure 2a).

2.2 Model Transformation and DPL

A model transformation $MT : M_1 \rightarrow M_2$ consists of a set of transformation rules t. Figure 4 shows a simple model transformation which contains one transformation rule. In the figure, the transformation rule $t : (\iota_P : P \rightarrow G(M_1)) \rightarrow (\iota_{P'} : P' \rightarrow G(M_2))$ is declared for a specific pattern P which in turn is an instance $\iota_P : P \rightarrow G(M_1)$ of M_1. For each match $m : P \rightarrow I$ of the pattern P in a given input instance $\iota : I \rightarrow G(M_1)$, $[\![t]\!]$ will produce a match $m' : P' \rightarrow I'$ of the pattern $P^{'}$ in the output instance $\iota^{'} : I^{'} \rightarrow G(M_2)$. That is, $[\![t]\!](m) = m'$.

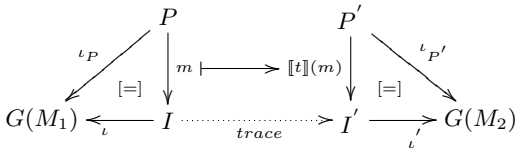
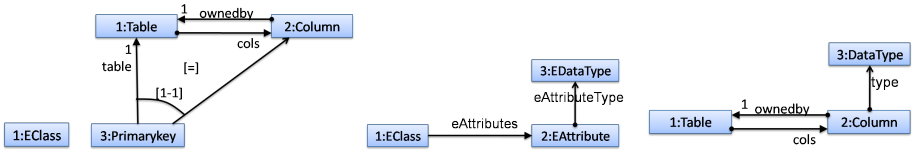


Fig. 4: A model transformation with only one transformation rule

For a match $m(P)$, the analogy is that P is the formal parameter of m and some part of I is the actual parameter. That is, a pattern P can be seen as a scheme and a match m assigns values from I to the variables in P. Moreover, we use $Match(P)$ to denote the set of all matches of P. Thus the input and output of transformation rules are set by patterns.

Figure 5 show examples of two transformation rules. The patterns P_1 and P_2 (Figures 5a and 5c, respectively) are instances of the metamodel of EMF (Figure 2a), and the patterns P_1' and P_2' (Figures 5b and 5d, respectively) are instances of the metamodel of RDB (Figure 2b). Matches of the patterns P_1 and P_2 are to be found in EMF models, i.e. instances of the Ecore model which is shown in Figure 2a. An example of such an instance (I_{EMF}) is shown in Figure 3. In this figure, the matches of the pattern P_1 are the nodes Person:EClass and Company:EClass. Moreover, the match of the pattern P_2 in I_{EMF} will be

.



(a) P_1 (b) P_1' (c) P_2 (d) P_2'

Fig. 5: The transformation rules t_1 and t_2.

Explicit Mapping between Patterns Generally in SE, equal names imply equal elements, e.g. the transformation rule t_1 in Figure 5 is understood as creating a `Table` for each `EClass`. To make this equality explicit, we introduce an explicit mapping of elements in the input and output patterns. This can be expressed by setting the requirement $P \xleftarrow{in_P} K \xrightarrow{in_{P'}} P'$ where $in_P; m = in_{P'}; m'$ (Figure 6). An explicit mapping between the rules in Figure 5 may be like $K_{t_1} := \{1 = 1\}$ and $K_{t_2} := \{1 = 1, 2 = 2, 3 = 3\}$. The set of mappings K_{t_1} are abbreviated such that:
$\{\, 1{:}EClass \xleftarrow{in_{P_1}} X \xrightarrow{in_{P_1'}} 1{:}Table \,\}$ with $X \mapsto 1{:}EClass$ and $X \mapsto 1{:}Table$ is written as $\{1 = 1\}$ where the first 1 is in P_1 and the second 1 is in P_1'[5].

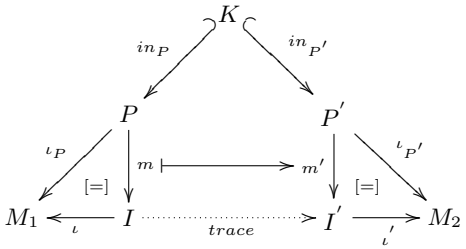


Fig. 6: A model transformation rule where the mappings between the patterns P and P' are expressed explicitly.

Thus, following our example of model transformation, the only match of the pattern P_2' that will be created in the target instance I_{RDB} (Figure 7) will be

.

Model Transformation vs Model Transaction It is important to distinguish between the concepts of model transformation and model transaction.

[5] The same abbreviation is applied for the coordinations $C_{i,j}$ and $C_{i,j}'$ (Definition 6)

While model transformation is a set of model transformation rules with a control mechanism to decide on the order of their execution on a transformation engine (operational), model transaction is the global declaration of the translation of an input model to an output model (declarative).

That is, model transaction $\mathcal{T}$: $M_1 \Rightarrow M_2$ is a mapping which independently of realization details specifies which instances of an input model has to be translated to which instances of an output model. In that sense, model transactions can be compared to transactions in database systems, where each transaction is composed of a series of database actions – queries and updates, executed in a specific order.

As an example of a model transaction, see the metamodels in Figure 2. The figures show only the declaration of the translation between the models without giving any details of the execution or the mappings between model elements. The model transaction in the figure is declared for the simplified versions of Ecore and the metamodel of RDB (Figures 2a and 2b respectively).

Definition 5. *A model transformation is a set* $MT = \{\ t_i, [\![t_i]\!]\}$*, where:*

- $t_i : Match(P_i) \rightarrow Match(P_i')$ *is a set of transformation rules' declaration*
- $[\![t_i]\!] : m_i \rightarrow m_i'$ *is the semantics of* t_i

The definition of model transformation above allows for heterogeneous model transformations. The corresponding homogeneous model transformation is defined in the same way for $M_1 = M_2$.

Coordinating Transformation Rules The way in which the set of transformation rules t_i in a model transformation are related to each other will depend on coordinations between the patterns of the rules. These coordinations are specified as follows.

Definition 6. *A coordination* $\mathcal{C}$ *for a model transformation* MT *is the sets* $C_{i,j}$ *and* $C'_{i,j}$*, where* $C_{i,j} := \{\ P_i \xleftarrow{in_{P_i}} C_{i,j} \xrightarrow{in_{P_j}} P_j\ \}$ *of common matches between the patterns* $P_i, P_j \in MT$ *for* $i \neq j$*, that is,* $in_{P_i}; m_i = in_{P_j}; m_j$*. The set* $C'_{i,j}$ *is defined in the same way for all* P'_i *and* P'_j *where* $i \neq j$*.*

The sets of coordinations between the patterns in Figure 5 may be defined as $C_{1,2} := \{1 = 1\}$ and $C'_{1,2} := \{1 = 1\}$. The coordination is a syntactical requirement for the alignment of the rules, i.e. to identify which elements of an input/output pattern is part of the other input/output patterns. This is necessary to avoid duplicates of elements in the generated model. A model transformation MT with a set of coordinations $\mathcal{C}$ is denoted by $MT^{\mathcal{C}}$.

The same model transformation MT with different sets of coordinations $\mathcal{C}$ may give rise to different model transactions, i.e. it may generate different instances of the output model. In the same way, different model transformations may create the same instance of the target model. This motivates for the design of a calculus to answer questions like: does a model transformation satisfy

a certain model transaction? and, if a model transformation satisfies a model transaction, are we able to prove that satisfaction?

Definition 7. *A model transformation $MT^{\mathcal{C}}$: $M_1 \Rightarrow M_2$ with a set of coordinations $\mathcal{C}$, satisfies a model transaction $\mathcal{T}$: $M_1 \Rightarrow M_2$, written $MT^{\mathcal{C}} \vDash \mathcal{T}$, if for each instance $\iota_1 : I_1 \rightarrow G(M_1)$ of M_1, $MT^{\mathcal{C}}$ generates an instance $\iota_2 : I_2 \rightarrow G(M_2)$ of M_2.*

The satisfaction condition above may be used as a condition for checking the correctness of model transformations. However, it's important to note that proving that a model transformation satisfies a model transaction may not be a trivial case. This discussion, which is part of our current research, will be detailed in a future work.

According to the model transaction $\mathcal{T}$ in Figure 2, any model transformation $MT^{\mathcal{C}} \vDash \mathcal{T}$ which satisfies $\mathcal{T}$, with the input model I_{EMF} (Figure 3), will cerate an instance of M_{RDB}, I_{RDB}, as shown in Figure 7.

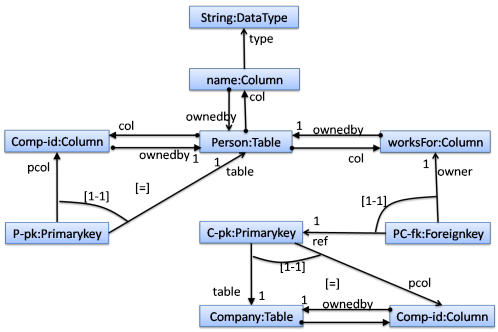


Fig. 7: The generated RDB model, I_{RDB}, which is an instance of the diagram specification M_{RDB} (Figure 2b).

3 Conclusion

In this paper we have given an overview of the state-of-the-art of our ongoing project on the formalization of modeling and model transformations based on the formal, diagrammatic framework DPL. We have argued that a formal, diagrammatic framework based on Category Theory is necessary for this formalization process and shown how models, metamodels, modeling languages, transformation rules, patterns and transformation definitions can be represented in DPL.

In DPL, each language L is represented by a formalism F_L which consists of a signature Σ_L and a metamodel M_L. The metamodel is specified as a Σ_L-specification if it is reflective, or as a Γ-specification where Γ is the signature of a common formalism which is used for the specification of the metamodels of the

languages in the DPL framework. L-models are then specified Σ_L-specifications. All Σ_L-specifications are required to both conform to M_L and be instances of M_L. We have distinguished between conformance to and being an instance of a (meta)model; as conformance is only concerned about the syntax of the models while being an instance means, in addition to conformance, respecting predicates (or constraints) that are set by the metamodel.

Model transformations in DPL are represented as a set of transformation rules which assign a match of an output pattern to matches of input patterns. The global declaration of a model transformation is distinguished from the operational description and called model transaction. Each model transformation is controlled through a coordination mechanism and the order of its rules. Thus the same set of rules may give rise to different model transactions. In addition, the same model transaction may be achieved by different model transformations. This result motivates for creating a calculus to reason about the satisfaction of model transactions by model transformations with different coordinations.

Some examples of models and model transformations are presented in the paper. However, the details of checking of conformance, checking validity of instances, checking satisfacition of model transactions and checking correctnett of transformations are not discussed in this overview paper.

References

1. Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *ICGT '02: Proceedings of the First International Conference on Graph Transformation*, pages 402–429, London, UK, 2002. Springer-Verlag.
2. Zinovy Diskin. The graph-based logic of ER-diagrams and taming heterogeneity of semantic data models, 1997.
3. Zinovy Diskin. *Mathematics of UML: Making the Odysseys of UML less dramatic. Practical foundations of business system specifications*, chapter 8, pages 145–178. Kluwer Academic Publishers, 2003.
4. Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: practice and promise*. Addison-Wesley, 1 edition, April 2003.
5. OMG. *OMG Model Driven Architecture Web Site*, June 2007. Object Management Group, http://www.omg.org/mda/index.htm.
6. Adrian Rutle, Uwe Wolter, and Yngve Lamo. Generalized sketches and model driven architecture. Technical Report 367, Department of Informatics, University of Bergen, Norway, 2008. Presented at CALCO Young Researchers Workshop 2007.
7. Uwe Wolter and Zinovy Diskin. The next hundred diagrammatic specification techniques: A gentle introduction to generalized sketches. Technical Report 358, Dept of Informatics, University of Bergen, July 2007.
8. Uwe Wolter and Zinovy Diskin. Generalized sketches: Towards a universal logic for diagrammatic modeling in software engineering. 2008. Proceedings, ACCAT 2007, ENTCS, Submitted.