Published as Chapter 8 in *Practical Foundations of Business System Specifications,*
Eds. Haim Kilov and Kenneth Baclawski. Kluwer Academic Publishers, 2003. 348pp.
ISBN 1-4020-1480-5

MATHEMATICS OF UML:
Making the Odysseys of UML less dramatic [1]

Zinovy Diskin
Universal Information Technology Consulting, Inc., *Detroit, USA*
Frame Inform System, Ltd., *Riga, Latvia*
zdiskin@acm.org

Abstract. It is commonly recognized that there are a few serious drawbacks in UML. Among the most often cited are (i) excessive size and complexity, (ii) limited customizability, (iii) unclear semantics. Less common is the understanding that these drawbacks are almost inevitable for UML due to the fundamental gaps in its logical foundations. Finally, it is barely known that the first of them instantly disappears, the second one becomes readily manageable and the third approachable, as soon as UML is based on a proper mathematical foundation already developed in mathematical category theory. One more, and very important, benefit is that categorical treatment of UML-models leads to a natural and simple notion of model mapping, and hence provides all the necessary prerequisites for efficient model management (cf. [BHP00]). In contrast, (iv) UML's model mappings are hardly definable in a manageable way due to (i) (and (iii) too), and it is a really serious problem still not recognized in the UML literature. The goal of the present paper is to manifest these claims, outline general mathematical mechanisms that do the job, and demonstrate how they work with simple examples.

> The craft of building information systems needs a shared language.... A language of these qualities may be a long way off, though UML2 will certainly make a step closer. The authors here debate the size and direction of this next step; another step will surely follow.
>
> *Joaquin Miller, "What UML should be", the Guest Editor's foreword to UML Panel in Comm. ACM [JM02]*

General landscape

UML is a *diagrammatic* language intended for specifying complex *abstract* structures of *diverse* nature. All the italicized words are essential, and jointly form a novel problem area where, it seems, the common engineering sense does

[1] The work was partially funded by the Latvian Council of Science.

not have much to offer. That is why UML developers are forced to invent a lot, from new concepts in semantics to new concepts in metatheory of modeling to notational tips, and grope their path through unfamiliar and obscure territories (the image suggested by the epigraph). Probably, the metaphor of Odysseys would not appear in the UML literature [Kob99] by accident.

Of course, such a situation is almost typical for engineering. Its long history is full of examples when an initial period of trials and errors finally led to success. However, in our case there is already developed a general map of the territory along with means facilitating the journey, why not use them to straighten (at least some) zigzags and avoid dead-ends? The history of engineering shows equally well that success stories in engineering areas are often connected with the use of suitable mathematical means. Not seldom, the latter had been already developed in mathematics in response to its own problems and, by some mysterious magic, turned out extremely suitable and helpful for applications.[1]

It appears that the relationships between software engineering (SE) and category theory (CT) are very much like the relationships between mechanical/electrical engineering and calculus. Indeed, diagrammatic specification of complex structures is not only a favorite problem in CT, but an everyday exercise of a working categorician (a mathematician thinking along CT-lines). For more than fifty years of managing the issue, CT developed a few fundamental ideas of great generality, created a useful working intuition and invented a convenient notation. It would be just reasonable to apply these ideas and notation to UML and see what from the CT arsenal might be useful.

The rest of the paper is organized as follows. Section 1 analyzes the interplay of syntax, semantics and logic for a modeling language, and argues that semantics is a key actor, without which the play collapses (or transforms into a quite different story). Section 2 applies these general considerations to a particular case with UML being the modeling language and arrow diagram logic developed in CT being the logic. The latter is based on a specification format called *sketch*, and the main thesis of the section along with the entire paper asserts an extreme usefulness of considering UML diagrams as visual presentations of sketches; section 2.4 briefly summarizes the benefits. Appendix contains more technical material. Section B presents three consequently richer formal frameworks where semantics for UML class diagrams can be built. Section D builds a behavioral formal semantics for *isPartOf* and composition. Section E outlines how an abstract mathematical theory of model management can be developed.

1 Semantics for a modeling language

1.1 Concerns usually attributed to "semantics" in the context of UML (or any other modeling language) can be separated in the following way.

Firstly, we have to figure out precise and detailed meaning of concrete modeling constructs of the language. For abstract and intuitively holistic concepts as, for example, *isA* and *isPartOf*, composition and aggregation of classes, be-

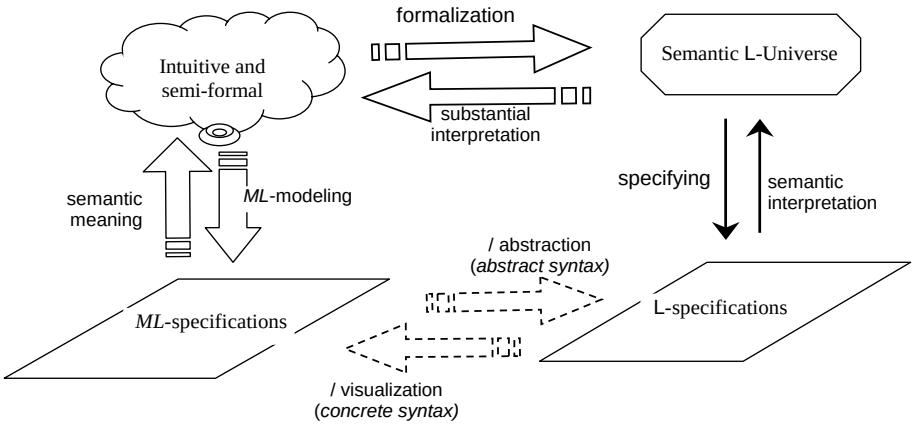


Figure 1: Formal logic for a modeling language: *ML* - a language, L - a logic.

ing "precise" means being formalizable, and formalization of such concepts is a highly non-trivial problem requiring special heuristic efforts.[2] It is an intellectual challenge for which there are no precise prescriptions of what and how to do. We could call this part of the entire semantic problem *substantial*.

Secondly, we have to deal with a much more technical and somewhat routine issue of specifying formal semantics we have found/built in a observable and comprehensible way. The question is what a formal language L we need to use so that intended L-semantics for L-syntax would conform to the formal semantic meaning we have found in the first stage of our efforts. Such a language L, including both syntax and intended semantics, is a logic in some general sense, and the second part of the entire semantic problem could be called *logical*.

Though these two parts can seem to be independent, actually they are intimately connected. A logic of specifying suggests certain patterns of reasoning and hence directs the heuristic efforts. Conversely, peculiarities of substantial semantics can fit well into specificational patterns of one logic and "do not want" to obey the rules of another logic – the resulting specification will be then awkward and unwieldy, and will resist substantial semantics search. In contrast, an appropriate logic leads to compact and elegant specifications and, as soon as it is found, everything begins to run smoothly. In this way, substantial semantics indirectly governs the choice of formal logic while the latter helps to solve heuristic problems in discovering/building the former.

The entire stage for this intriguing play is schematically presented on Fig. 1. Note the two lower horizontal arrows that are derived by composition of other arrows in an evident way. These arrows mutually relate specifications in the modeling language (written in some concrete syntax) with their underlying logical counterparts (hidden in the former and written in some abstract syntax).

1.2 In a healthy modeling language *ML*, its notational organism (left side of the schema) is built upon a solid logical skeleton determined by formal seman-

tics of *ML*'s constructs (right side). In contrast, the lack of formal semantics destroys the chain and deprives us of reasonable foundations in the choice of specificational logic and syntax. The *ML*-syntax is then necessarily extracted from the intuitive semantics and follows all the bends and twists of the intuitive perception. It gives rise to a "congruently" bulky vocabulary of basic concepts, with unmanageable overlapping and unintentional synonymy and homonymy. Other factors influencing the choice of syntax are conformity to notational traditions in the domain, common engineering sense and subjective preferences of language designers. Being normal and healthy in general, *in the absence of the right side of the schema* these factors can only contribute to the mess rather than order it. On a whole, we come to an arbitrary syntax and arbitrary (if any) specification logic, which are based on intuition (necessarily fuzzy) and free notational taste (necessarily subjective). The entire phenomenon might be called *lack-of-semantics syndrome*, and it seems that many symptoms of UML's not-well-being fit into this syndrome.

If the diagnosis above is correct, then an actual rather than symptomatic treatment should begin with putting UML on proper semantic and logical foundations, which, as it was discussed above, are normally to be built together. It's a complex problem, the more so on the UML's scale of all-embracing modeling language. To do this job, we need to build formal semantics for all, or at least, for all major constructs in UML along the lines of the schema on Fig. 1.

1.3 An essential part of the formalization work was fulfilled in [DK03, Dis02a], where a formal semantics for UML class diagrams is built (fragments can be found in [Dis00, Dis02b]).[3] In accordance with a general unifying spirit of UML, all the constructs are formalized within the same specification logic developed in CT. This logic is based on predicates over arrow diagrams, hence the name: *arrow diagram logic*, ArrDL (presentation of ArrDL as an immediate graph-based generalization of string-based first order logic can be found in [Dis97]). This logic is absolutely expressible (sect. 2.1.2) and hence all constructs in structural, behavioral and meta-modeling sections of UML can be specified within the same syntactic format of ArrDL. The format is called *sketch* (the term is also borrowed form CT), and we will also use the terms *sketch logic* or *sketch modeling language*, SkeML. Correspondingly, the 'abstraction' arrow of general schema Fig. 1 turns into a procedure of *sketching* UML diagrams, while the latter appear as visual presentations of sketches. The entire schema on Fig. 1 then gives rise to a general schema of sketching UML-diagrams, Fig. 3 on p.151, where, for now, do not mind the internal structure of the top right node.

The sketch-based formalization of UML class diagrams provides a very productive feedback for UML. It suggests a few basic notational concepts missed in the UML, e.g., those of diagram predicate and diagram operation. It cleans up the vocabulary of modeling constructs: clarifies the meaning of the existing constructs, splits them into a few independent sub-constructs if necessary, adds a few novel constructs (without complicating the syntax !) – some samples of this work are presented in Appendixes C, D.2, D.3 and in [Dis02b, DK03, Dis02a].

Finally, it provides all the prerequisites for integrating the UML's parts into a single language, whose multiple subnotations are based on the same syntactic format and have formal semanticses embedded into a unified formal framework.

2 Mathematics of UML: UML diagrams as visual abbreviations of sketches

2.1 Sketches and their interpretations

2.1.1 Getting started. CT is full of fairly complex structures, and they all are specified in the same unified format called *sketch*. Briefly, a sketch is directed multigraph (that is, there are possible multiple arrows between the same two nodes), in which some diagrams (fragments of the graph closed in some technical sense) are marked with predicate labels taken from a predefined signature.[4] A few simple sketches are presented on Fig. 2 where diagram markers are shown in square brackets. In all sketches on Fig. 2 the diagram of a marker is the entire graph, but it is a very special situation and normally a marked diagram is just a fragment of the graph. An example – a simple sketch specifying the extensional side of semantics for a simple UML class diagram can be found in Appendix A.3.

So, a sketch consists of only three sorts of items: nodes, arrows, marked diagrams. The latter are to be understood as statements, or predicate declarations, about nodes and arrows occurring into the diagram. For example, on Fig. 2 each sketch presents a statement about all its items considered together. More accurately, each predicate label P has its *arity shape* – a graph D_P (or a family of graphs), and can mark (be hung on) only diagrams of shape D_P. This is a syntactical rule. Semantically, if some universe $\mathbf{U}$ for intended interpretations of sketch's nodes and arrows is given (which is always the case in engineering applications), then P denotes a certain property, $P^{\mathbf{U}}$, of configurations in $\mathbf{U}$ of shape D_P. The use of sketches is based on formal semantics going first, before syntax, and so predicate labels legally used in sketches must have a formal meaning.

For example, consider the *sets-and-mappings* interpretation for sketch nodes and arrows, when $\mathbf{U}$ is some universe of sets and mappings, **Set**. Then predicate labels denote properties of sets-and-mappings configurations, for instance, the property of two mappings going into the opposite directions to be mutually inverse, Fig. 2(a). Or, for the *states-and-transitions* interpretation of nodes and arrows, predicate labels denote properties of configurations of states-and-transitions, for instance, the property [`prod`] of a state concurrently include a few other states with the corresponding *projection transitions* that do nothing besides observing the component of the complex state, Fig. 2(b). Or, say, for a *database dynamics* interpretation where nodes of a sketch denote database states while arrows are transactions, predicate labels denote properties of configurations of database states and transactions; for instance, a special property of an ordered pair of transactions going into the opposite directions (Fig. 2(a)

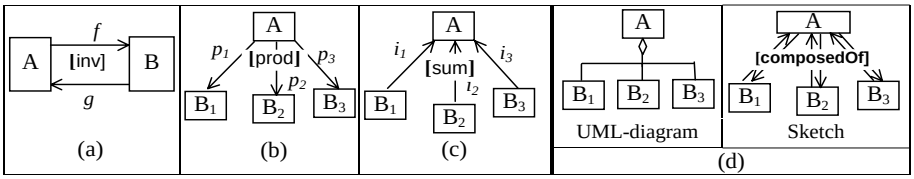


Figure 2: Several simple sketches.

again), when the second transaction annuls all the changes made by the first transaction (roll-back). Or, else, nodes of a sketch may denote UML models (or ER-diagrams, or sketches) and arrows are model mappings, then a diagram predicate may denote the property of a model to be the merge of other models occurring into the diagram with the corresponding *inclusion mappings*, Fig. 2(c). Section A in Appendix accurately specifies the sketch syntax and presents examples of how sketches can be used for specifying.

2.1.2 Expressiveness of the sketch language. The brief outline above shows a great flexibility of the sketch language, but what about its expressiveness? That is, how many constructs and artifacts of software engineering, which UML is intended to model, can be expressed in the sketch language, *ie*, by nodes, arrows, diagram predicates? It is indeed a crucial question, and CT gives a somewhat surprising answer: any construct that can be specified formally, can be specified by a sketch as well (in a suitable signature of diagram predicates).

Speaking technically, what was proved in CT is that formal set theories currently known in mathematics (Zermelo-Frankel, type theory, and others) are interpretable in the sketch language, or *sketchable*. However, the very notion of formalizability of a construct means nothing but its expressibility in some set theory, that is, owing to the result just mentioned, expressibility by a sketch. Moreover, the number of arrow diagram predicates necessary to express formal set theories is fairly small, about ten or so (it depends on what set theory we need to sketch and how some CT-notions involved are formalized). Thus, any formal construction can be specified by a sketch in a fixed signature of diagram predicates. What this result means for UML, and how sketches can be applied to UML, is discussed in the next section.

2.2 Sketching UML diagrams

A general schema of connections between UML diagrams and categorical sketches is presented on Fig. 3. We start with an UML diagram D having, let's suppose, a clear (yet informal) semantic meaning $M_{inf}[\mathsf{D}]$. Then we formalize it in terms of abstract sets and mappings and come to a formal explication/ formal refinement $\mathsf{M}_{\mathsf{frm}}[\mathsf{D}]$ of $M_{inf}[\mathsf{D}]$ (still do not mind the internal structure of the upper right node). Finally, we specify $\mathsf{M}_{\mathsf{frm}}[\mathsf{D}]$ by a sketch S and call the entire procedure *sketching diagram* D (one of the lower horizontal arrows).

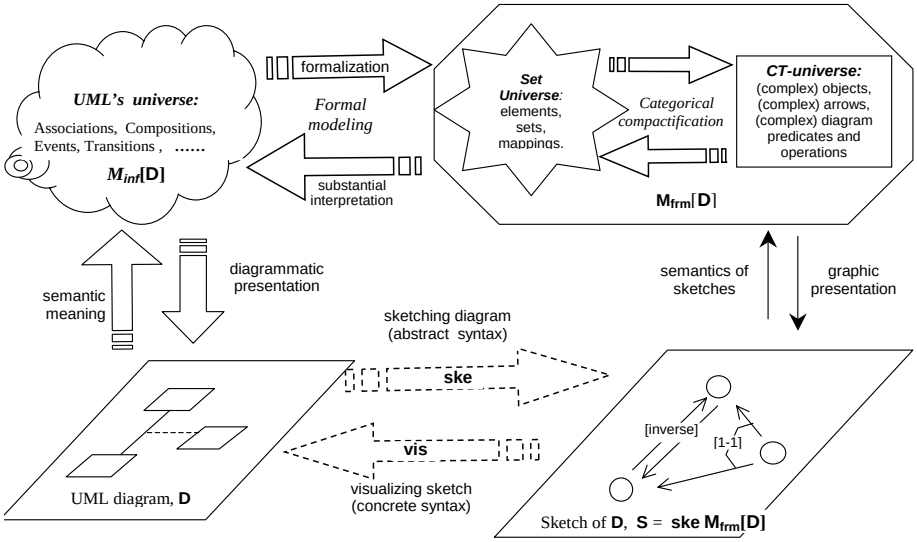


Figure 3: Sketching UML-diagrams: general schema

2.2.1 The key problem in the entire procedure is the *formalization* arrow. The absolute expressiveness of the sketch language says that this arrow exists as soon as at least some formalization exists, but it says nothing how to find the latter. This is the most subtle point of sketching UML diagrams, which can require non-trivial heuristic efforts. Let's suppose however that we have overcome the severe difficulties of formalization and built a formal semantics for a class of UML modeling constructs, and then expressed it by the corresponding sketches. What real benefits, besides those provided by just having a formal semantics, can it bring to UML?

A peculiarity of sketch-based formal semantics is that formal specifications are also diagrammatic, and we can hope to find some useful correlations between the two graphical images – an UML diagram D and a sketch S, specifying D's formal semantic meaning. For example, UML's activity diagrams look almost like sketches from the very beginning. They have nodes and arrows between them, while some additional constructs like *fork* and *join* of arrows actually denote diagram predicates of shapes shown on, respectively, Fig. 2(b,c). Statechart diagrams have complex node substructures but they can be made much closer to sketches if we rework presentation of composite states (nodes) in the following way. Component nodes are removed from the frame of the composite node, and then the arrows connecting the composite with the components are added: *projections* for concurrent composition (Fig. 2b) and *inclusions* for sequential (Fig. 2c). As for UML's sequential diagrams, it seems they might be converted into sketches, enriched with a linear (or, more generally, partial) order on their arrows (a favorite topic in CT).

Of course, this is only a very rough outline. Building complete formal models

for UML's activity/statechart/sequential diagrams may require adding many additional details so that resulting sketches will not be so immediately close to their UML diagram preimages.

2.2.2 The situation may look even more problematic for sketching UML's class diagrams. Extracting their carrier graphs is more or less clear: classes are nodes and binary associations are pairs of mutually inverse arrows between them; as for n-ary associations, Appendix C shows how to manage them via arrows and diagram predicates. However, UML class diagrams contain also association links with special semantics like composition, aggregation, generalization and others. We can hope to model them by the corresponding diagram predicates, for example, as shown on Fig. 2(d) for UML's composition, but here we come to a crucial point.

To use the marker [`composedOf`] on Fig. 2(d) in a legal way, we have to know its formal meaning. Figuring out the latter is a non-trivial task, but let's suppose we have accomplished it, that is, built a formal model for the notion of *isPartOf* and composition. The experience of building formal models shows, however, that the more construct to be formalized is intuitively evident (e.g., *continuous* curve or *isPartOf*), the more intricate is its formalization, and the more number of ingredients (sets, mappings, relations, operations) is involved. This aspect becomes especially essential in formalizing engineering constructs and artifacts, and often makes their formal models bulky and hard to comprehend, to operate and to reason about.[5] It often becomes a stumbling block for industrial applications of formal models (the "nightmare" of engineers trying to use them and the "evil" of theorists trying to install them in industry). This point is illustrated on Fig. 3 by a special thorny shape of the node of immediate formalization. The sketch language can help here owing to its graphical format, but if the immediate formal model involves a lot of sets and mappings, then its sketch will be just an unwieldy graphic specification (e.g., the sketch of immediate formal model of composition is far not so simple as is shown on Fig. 2d).

2.2.3 Fortunately, the sketch language brings to the area of formal modeling much more that just a graph-based specification format. The necessity to manage really complex abstract structures and work with them (write them out, manipulate them and reason about them) is an everyday exercise of a working categorician. A useful experience of finding compact presentations of formal constructions was accumulated in CT, and an effective system of the corresponding abstract notions was developed. As a rule, such notions are special mappings so that CT is full of arrows encoding complex constructions and structures. In this way bulky specifications are packed into compact arrow diagrams. (Appendixes D and E present samples of how it works in two important areas: specifying object behavior and model management; many complex issues in these areas are greatly clarified and simplified by using arrow specifications). It is this well elaborated abstraction and modularization mechanism, along with diagrammatic syntax, makes categorical logic preferable in many tasks of formal modeling in comparison with other formalisms from FOL to Z

to ASM (Abstract State Machine, see [BÖ2] for a survey) and their versions.

Thus, we assume that in the sketch realization of our general schema Fig. 1, the right top node includes some compactification procedure crucial for engineering applications: immediate (bulky and "thorny") formal semantics is packed into compact arrow patterns and only then is specified by a sketch as shown on Fig. 3. As a result, an UML diagram D appears as a pair (S, V), where S is a sketch specifying D's formal semantics and V is a visual presentation of S in the form of UML-diagram.

2.3 UML diagrams as visual abbreviations of sketches

Considerations of the previous section may look more or less reasonable, but they will remain only hypothetical premises until they are applied and do the job. And it was really done for sketching UML class diagram including semantic associations between object classes (*isA*, *isPartOf*, composition and aggregation, specialization and generalization), see [Dis02b, Dis02a, DK03]. Briefly, the idea is as follows.

2.3.1 Semantics of an object class has two sides: extensional – an object class is a collection of objects, and behavioral – objects have changeable states. The extensional side is modeled by the construct of *variable* set (varset), that is, a sequence of sets indexed by time moments and interconnected by binary relations explicating object identity. The behavioral side is captured by assigning to a class some *coalgebra* – an elegant categorical counterpart of the notion of state machine.

The construct of *behavior set* (behset) combines extension and behavior in some coherent way. A behset consists of a varset, a coalgebra (state machine) and a mapping of the former into the latter (Appendix B presents details).This framework allows adequate formalization of all the constructs used in UML class diagrams: object classes are behsets, binary associations are pairs of mutually inverse mappings between them, n-ary associations and semantic relationships between classes amount to certain diagram predicates over the corresponding configurations of behsets and mappings between them.

So, thinking semantically, a system of classes and associations gives rise to a system of behavioral sets and mappings, over which some diagram predicates are declared. Such a system is well specified by a sketch: nodes of the sketch denote behsets, arrows are mappings between them and marked diagrams are properties of behset configurations. Note, each item in the sketch encodes a fairly complex arrow diagram (see, for example, Fig. 6 on p. 163, where semantic interpretation of a single node is shown). Finally, it turns out that for many constructs used in UML class diagrams, their sketches are graphically similar to their UML presentation. It gives rise to a general graphical similarity between an UML diagram D and the sketch S in D's sketch presentation (S, V).[6]

2.3.2 For some modeling constructs, e.g., binary association or generalization, their UML presentation is so much similar to sketches that UML diagrams can be well considered as visual abbreviation of sketches. This viewpoint allows an-

alyzing UML's syntax on a precise foundation. Namely, we can try to organize notational units used in UML and SkeML into similar mathematical structures and then consider visualization V as a morphism of these structures. Diagram D is then appear as the image of sketch S under this mapping V. Very likely, general patterns of algebraic semiotics [Gog98] will work well here, see [Dis02b] for some details.

There are also constructs whose UML's presentation is deficient: it obscures semantic picture or misses some semantically valid elements at all. For example, Appendix C demonstrates how a technically very simple idea to specify an n-ary association by a diagram predicate at once solves the infamous problem of multiplicities for n-ary associations. Also, it is shown in [Dis02b] how semantically transparent and syntactically elegant are the sketch specifications of qualified associations. They become especially preferable in comparison with UML's notation, when additional constraints to qualification are considered. The reader is strongly encouraged to look through examples in [Dis02b].

Thus, seen from the viewpoint of intended semantics, UML syntax appears as just a system of special visualizations of arrow diagram predicates; sometimes they are apt to be accurate visual abbreviations, but sometimes they are awkward and ambiguous (but see a brief discussion in Conclusion).

2.4 Sketches vs. UML diagrams: Summary of benefits

2.4.1 Size and Complexity. The SkeML vocabulary of basic concepts consists of only four items (node, arrow, diagram predicate and operation) vs. about seventy in UML. Correspondingly, a sketch specification of SkeML metamodel [Dis02a] consists of seven nodes and eleven arrows vs. UML metamodel containing more than one hundred metaclasses.

2.4.2 Customizability. Any sketch is a sketch in a predefined signature of diagram predicates and operations. It gives rise to a quite immediate customization mechanism for sketches: a given class of users defines its own signature convenient for its modeling tasks. When we pass from sketches to UML, one more customization parameter occurs: visualization of sketches in a given signature. It seems that combination of these two – signature and visualization – will provide enough flexibility for customization and extension mechanisms. Note also that because of the absolute expressivity of the sketch language, *any* customization/extension can be captured by sketches as soon as we know its formal semantics. This feature could really bring a peace of mind for tool vendors.

2.4.3 Semantics and logic. Generally speaking, the question of what is the logic of UML seems not even stated in the UML literature. UML's huge syntactical apparatus is built on somewhat arbitrary and *ad hoc* foundations (of previous notations for OO modeling that were also built in an *ad hoc* way) beyond strict logical patterns. This logical arbitrariness is one major cause of UML's excessive size and complexity. The other one is absence of clear semantics for a majority of modeling constructs (that is well recognized). As it was

discussed in sect. 1, these two gaps are closely interrelated and simultaneously covered as soon as we have built formal semantics for UML's constructs and specify it graphically by sketches, it was said enough about that in the paper, see 2.2, 2.3. It should be stressed (once again) that the final approval or rejection of the sketch language as an *adequate* specification foundation for UML must be based on the consideration of concrete examples of sketching UML diagrams. These examples can be found in [DK03, Dis02b, Dis02a]; hopefully, they are more than convincing.

2.4.4 Model management (MMt). Excessive size and complexity of UML models directly lead to complexity of their management, and can make it really hard. A typical example is an excessive size of XMI(= XML Metadata Interchange format) – a recent OMG's standard for model interchange. For another example, consider the notion of model mapping that should be among the major ones in MMt, see argumentation in [BHP00] and Appendix E.[7] However, UML model mappings could be hardly defined in a manageable way due to a bulky repertoire of modeling constructs used in UML, even within a given UML's sublanguage.

In contrast, the sketch treatment of UML models can greatly facilitate their management. A general prerequisite is that as soon as UML-diagrams are considered as visualizations of sketches, UML model management can be separated into sketch management and visualization management, which itself can ease MMt. Moreover, in many MMt tasks, e.g., model interchange, the second component is at all redundant: visualization is done on the site only when we need to present/view the model. As for sketch MMt, it is greatly facilitated by extreme compactness of sketch syntax and its graph-based nature. Particularly, the advantages of SkeML are straightforward for the following issues in MMt.

Model serialization and interchange via XML. Since sketches are just labeled graphs with some extra labeling on diagrams, their XML-serialization is straightforward and compact.

Model mappings and repositories. The notion of mapping between sketches is natural and simple: it is a mapping of underlying graphs compatible with marked diagrams, see Appendix A.4. It at once makes the machinery outlined in Appendix

Managing models' heterogeneity. When the model repository consists of models of different types, the corresponding sketches will be sketches in different signatures. However, sketches in different signatures are nevertheless sketches, *ie*, specifications in the same format, and they can be uniformly compared and related via relating their signatures, see A.4.3(c).

Reflexivity. On the level of modeling, a repository of models (sketches or others) can be specified by a *model sketch* whose nodes denote models and arrows denote model mappings. Marked diagrams in such a sketch denote statements about models, and manipulations with models are diagram operations over this sketch. On the level of metamodeling, the sketch metamodel is very compactly specified by a sketch. Any sketch signature is also specified by a

sketch whose nodes denote diagram predicates (markers) and arrows denote logical entailment between them. A repository of signatures is then specified by a *signature sketch*, and manipulations with signatures can be specified by diagram operations over it. Thus, managing the entire heterogeneous model repository on all its levels and meta-levels can be based on the same sketch format. If models are sketches, then there is nothing in the world of models, including heterogeneous model management and metamodeling, besides sketches and their mappings.

3 Conclusion

This paper presents mathematical foundations for UML built within a graph-based specification language of *sketches*, SkeML; the term and basics of the machinery are borrowed from mathematical category theory. It was manifested and, in part, demonstrated, that SkeML provides an extremely powerful specification framework capable to serve well all areas of OO modeling: business/domain modeling itself, metamodeling and model management, and provides effective customization, extension and modularization mechanisms. All these jobs are done within the same compact syntax (just nodes, arrows, diagram predicates) to which different semantic interpretations are assigned in function of the area of application ("plug-and-play"semantics). The entire framework, from syntax to semantics, is well layered and modularized, and can be based exclusively on the notion of sketch and sketch mapping (total reflexivity).

In terms of the (3C)-proposal for UML – *Clear/Clean/Concise UML* (see [JM02] for a brief outline), SkeML proposal can be encoded by (3EC) – Extremely Clear/Extremely Clean/Extremely Concise. On the other hand, the arrow formalization of metamodeling, and availability of formal semantics for sketches, makes them an ideal specification base for realizing a dream of the (2U)-proposal for UML (*Unambiguous UML*, see [JM02] again) to have a complete tool chain from ModelBuilders to ModelTesters.

Despite these strong advantages of SkeML, relations between UML diagrams and sketches are much more delicate and intricate than just the alternative of which notation to use. Indeed, the sketch format is more like a specificational skeleton while UML is a full-blooded notation developed to serve real engineering needs. UML diagrams must conform to notational and terminological traditions in the domain, and provide enough auxiliary and extra-specificational means of facilitating the practical use of the diagrams. The latter is often called "syntactic sugar", but as it was noted in [Gog93], "discussions about notation can be more heated and protracted than discussions about issues usually considered more substantive. This suggests there may be more to notation than is admitted in phrases like *mere syntax* and *syntactic sugar*".

On the other hand, sketches need some means of visual presentation. Then UML and SkeML can be conformed rather than confronted, if we consider *UML diagrams as a means of visual presentation of sketches*. It would also provide

UML with an important quality of clear separation of specification (sketches) and visualization (UML diagrams), a requirement specially emphasizes in the (2U)-proposal. Particularly, it will give one more dimension for adaptability and customizability of UML models, and will greatly facilitate model interchange. And some pieces of sketch language, for example, the notion of diagram predicate and operation, or the arrow diagrammatic specification of n-ary associations and qualified associations, can be immediately incorporated into UML.

On a whole, the call to set OO visual modeling on the sketch/ArrDL foundation, and then to view UML diagrams as visual abbreviations of sketches, should change the understanding of UML notation rather than notation as such. This view provides a way of evolutional rather than revolutional development of UML, which is strongly suggested by the group of *UML2.0 Partners* (see [JM02]), and leads to a balanced treatment of UML's problems. This treatment assumes some partial surgery, but is mainly therapeutical and, as we have seen, naturally integrates many features declared to be beneficial for UML in various proposal on UML development. At any rate, the sketch proposal aims at treating the illness rather than symptoms, and sets up a healthy skeleton for UML's notational organism and its future development.

Acknowledgement. Most of all I am indebted to Haim Kilov for multiple discussions and general support of the very idea of abstract mathematics' intervention in the OO visual modeling, for his proposal to transform (a part of) these discussions into a paper, and finally for his endless editor's patience during endless reworking of the paper. Haim's comments and merciless yet precisely pointed critique essentially influenced the content, shape and structure of the paper. If they are still not good enough, it is because of my stubbornness rather than the editors' overlook. The ideas presented in the paper were also discussed with Bran Selic, in general and in details, and with Joaquin Miller, in an overall context of mathematics (and CT) vs. engineering (and SE). I am grateful to them for helpful comments and encouragement (coming from the industrial side, it is especially important for a mathematician daring to write about industrial standards). Discussions with Ken Baclawski and Bernhard Rumpe (and encouragement from the computer science side) are also gratefully appreciated. Special thanks go to my colleagues at F.I.S., Boris Kadish and George Sheinkman, for multiple discussions of the subject, which started long before UML appeared on the stage. Finally, maybe I should also express a mathematician's gratitude to the UML's creators and developers, whose really heroic job allowed shaping a somewhat fuzzy subject of mathematics of OO visual modeling into a precise discipline of *mathematics of UML*.

Appendix

A Arrows, diagram predicates, sketches

A.1 Some notation and terminology about arrows.

Given sets X,Y, we assume a general mapping between them $f\colon X \to Y$ to be partially-defined and multi-valued. The *domain* of f is the set $Dom(f) \subseteq X$ of

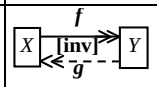
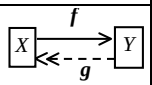
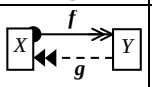
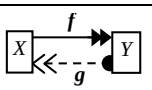
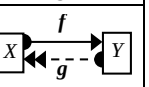
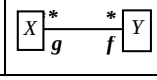
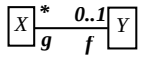
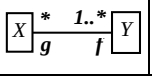
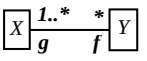
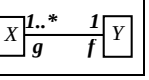
Seman-tics	f is partial and multi-valued, g is its inverse	f is single-valued	f is total (hence, g is covering)	f is covering (hence, g is total)	f is a total single-valued cover
	1	2	3	4	5
Sketch notation					
UML notation					

Figure 4: Properties of a general mapping $f\colon X \to Y$, the arrow g and its properties are derived.

all $x \in X$ for which $x.f$ is defined, and if $x \in Dom(f)$ then $x.f$ is a non-empty subset of Y. Here we write $x.f$ for the value of f at the argument x, we will also write $f(x)$ and sometimes $f.x$ for that (it will be always clear from the context what is the mapping and what is the argument). If f is *a priori* known to be single-valued then we consider $x.f$ an element of Y rather than a singleton subset of Y. The sets X and Y will be called the *source* and the *target* of f and denoted by $\Box f$ and $f\Box$ respectively. We call an arrow f *totally-defined* or just *total* if $Dom(f) = \Box f$. Dually to the notion of domain, the *range* of f, $Rg(f)$, is defined to be the set of all $y \in Y$ s.t. $y \in x.f$ for some $x \in Dom(f)$. Sometimes we will also write $f(X)$ for $Rg(f)$ and call it the *image* of X under f. We will call f a *cover* if $Rg(f) = f\Box$, this property is dual to being totally-defined.

A single-valued mapping is called *injective* or *1-1*, or else *monic* (a term used in CT) if for any two different elements $x, x' \in Dom f$, $x.f$ and $x'.f$ are also different.

To denote these properties of mappings in our diagrams, we need some consistent graphic notation, a suitable variant is presented in Fig. 4. Its advantage is that a combination of arrow properties is visualized by the corresponding combination of visualizations, an example is in column 5.

If we have two single-valued mappings $f\colon X \to Y$ and $g\colon Y \to Z$ such that $f\Box = \Box g$, then f and g can be composed into a mapping $(f \diamond g)\colon X \to Z$ defined in an evident way: for any $x \in Dom(f)$, $x.(f \diamond g) \stackrel{\text{def}}{=} x.f.g$ if $x.f \in Dom(g)$, and undefined otherwise.

An *arrow diagram* (or just *diagram*) is a configuration of nodes (in this paper, usually denoting sets) and arrows between them (mappings). A diagram consisting of two nodes X, Y and two arrow paths between them,

$$X \xrightarrow{f_1} A_1 \dots A_m \xrightarrow{f_m} Y \text{ and } X \xrightarrow{g_1} B_1 \dots B_n \xrightarrow{g_n} Y,$$

is called *commutative* if $f_1 \diamond \dots \diamond f_m = g_1 \diamond \dots \diamond g_n$. That is, for any $x \in X$, equality $x.f_1.\ \dots\ .f_m = x.g_1.\ \dots\ .g_n$ holds.

A.2 The construct of key: A key construct missed in UML

In this section we will consider a simple diagram predicate for the universe of sets and mappings. It should be one of the major in OO modeling since it determines a way of objects' identification. In fact, it is a diagram counterpart of the notion of key to

relation well known in the relational data model. Implicitly, this predicate is of course present in UML diagrams, but it is hidden in various visualizations. Our goal is to make it explicit; as we will see below in sect. C, it greatly clarifies structural aspects of n-ry associations' semantics.

Let $f_i\colon\ X \to Y_i, i = 1, \ldots, n$, be a family of total single-valued mappings with the same source, or a *span*. We call such a span *jointly one-one (monic)* if it satisfies the following condition:

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

In other words, the family of mappings f_i jointly identifies elements of X in a unique way. Then the tuple-mapping $f = [f_1 \ldots f_n]$ into the Cartesian product of target sets,

$$f = [f_1 \ldots f_n]\colon\ X \longrightarrow Y_1 \times \cdots \times Y_n, \quad x.f \stackrel{\text{def}}{=} [x.f_1, \ldots, x.f_n]$$

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

Since, conversely, for any relation the family of its projection mappings is jointly monic, the very notions of relation and a jointly monic span are equivalent. Correspondingly, on the syntax level, to specify a node as a relation one may leave the node without any marking but label instead the corresponding span by marker [1-1]. In fact, this is nothing but a well known idea of designating a key to relation and we could call a monic family of arrows also a *key* to its source.

A.3 Sketches informally, an example

A simple example of sketch is presented on Fig. 5, on the right. The sketch consists of three nodes and four arrows that denote sets and corresponding mappings between them; mappings are partially defined and multi-valued unless some constraints are declared for them. Besides the carrying graph, the sketch contains a few marked diagrams, that is, fragments of the graph marked with predicate labels: the pair of arrows (*employs*,*worksFor*) is marked with label [inv], the arrow span (*employer*,*employee*) is marked with label [1-1] and the entire triangle diagram is marked by label [graph]. (Sources and targets of arrows occurring into a diagram are always also included into the diagrams). In addition, figurative heads and tails of arrows also denote predicate labels hung on (diagrams consisting of single) arrows. (Figuring out the diagram of some predicate label in a sketch drawn on paper might be problematic, but when sketches are drawn on a computer display, it is quite easy: clicking the label highlights the diagram).

Each of the labels is taken from a predefined collection, a *predicate signature*, and denotes a certain property of a diagram on which it is hung. Meaning of arrow heads and tails is explained on Fig. 4. The label [inv] states that the two mappings must be mutually inverse. The predicate [1-1] (sect. A.2) states that the pair of mappings (*employer*,*employee*) jointly provides one-one identification of *Job*-objects. It implies that *Job*-objects can be identified with pairs (C, P) with C a *Company*-object and P a *Person*-object, that is, the set *Job* is a binary relation over sets *Company* and *Person*.

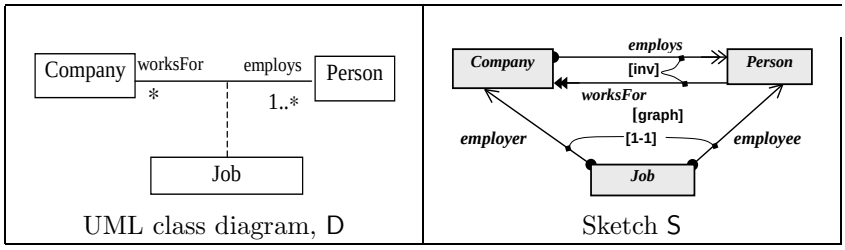


Figure 5: Sketching UML diagrams: an example.

The marker [graph] hung on the triangle diagram states that this binary relation is nothing but the graph of (either of) the two mappings in the triangle's base.

As for relation of this sketch to the UML diagram on the right of Fig. 5, the sketch specifies just a poor projection of the semantic picture – that one which can be expressed within the framework of static sets. To capture richer aspects of semantics, we need to interpret the sketch items in a richer universe of variable sets or even behavioral sets (sect. B). However, a richer semantic interpretation doesn't affect the syntax! What will be changed is the meaning of nodes, arrows and markers, and maybe a few new markers (senseless or trivial for the static set universe) will be added, but the carrying graph remains the same.

A.4 Formal definitions

A.4.1 Graphs. (a) A *(directed multi)graph* is a collection of nodes and arrows between them. There may be multiple arrows between the same two nodes.

(b) A *graph morphism or mapping* $f\colon G_1 \to G_2$ is a mapping of nodes to nodes and arrows to arrows such that their incidence is preserved. In a certain context, graph mappings are called *functors*. Namely, when graphs' arrows can be composed and a graph mapping preserves composition, it is a functor.

(c) A *diagram* in a graph G is a graph mapping $d\colon G_d \to G$ with G_d a graph called the *shape* of d. Informally, d often may be considered as the image of this mapping – a subgraph $d(G_d)$ of G.

A.4.2 Signatures. (a) A *(diagram predicate) signature* is a set Σ of *predicate labels* together with assignment to each label $P \in \Sigma$ its *arity shape* – a graph (or a family of graphs), $Arity(P)$.

(b) A *signature morphism* is a mapping $\sigma\colon \Sigma_1 \to \Sigma_2$ such that $Arity(P.\sigma) = Arity(P)$ for every $P \in \Sigma_1$.

A.4.3 Sketches. (a) Given a signature Σ, a *(Σ-)sketch* S consists of a graph G_{S}– the *carrier* of the sketch, and, for every label P from Σ, a set (maybe, empty) $D_{\mathsf{S}}(P)$ of diagrams in G_{S} *marked by* P.

Formally, a marked diagrams is just a pair (d, P) with P a predicate label and $d\colon G_d \to G_{\mathsf{S}}$ a diagram in G_{S}. Of course, shapes of diagrams marked by P must be isomorphic to $Arity(P)$ (or belong to the family $Arity(P)$ if the P's arity is a family of shapes).

(b) A *sketch morphism or mapping* $f\colon \mathsf{S}_1 \to \mathsf{S}_2$ is a mapping of the carrier graphs, $f\colon G_{\mathsf{S}1} \to G_{\mathsf{S}2}$, compatible with marked diagrams in the following way. If a diagram $d\colon G_d \to G_{\mathsf{S}1}$ is marked by P in S_1, $d \in D_{\mathsf{S}1}(P)$, then its image in $G_{\mathsf{S}2}$, the diagram $d \mathbin{\triangleright} f\colon G_d \to G_{\mathsf{S}1} \to G_{\mathsf{S}2}$ must be also marked by P in S_2, $d \mathbin{\triangleright} f \in D_{\mathsf{S}2}(P)$.

(c) Given a signature Σ, let **Sketch**(Σ) denotes the space of all Σ-sketches and their mappings. Evidently, a signature morphism $\sigma\colon\ \Sigma_1 \to \Sigma_2$ gives rise to a mapping between sketch spaces, $\sigma_{\#}\colon$ **Sketch**(Σ_1) $\to$ **Sketch**(Σ_2).

Now we can define mappings between sketches in different signatures, that is, pairs (Σ, S) with Σ a signature and S a Σ-sketch. Namely, a mapping $\phi\colon\ (\Sigma_1, \mathsf{S}_1) \to (\Sigma_2, \mathsf{S}_2)$ is a pair (σ_ϕ, f_ϕ) with $\sigma\colon\ \Sigma_1 \to \Sigma_2$ a signature morphism and $f_\phi\colon\ \mathsf{S}_1.\sigma_{\#} \to \mathsf{S}_2$ a mapping of Σ_2-sketches.

A.5 Semantic sketches

Sketches we consider are not necessarily syntactic constructs that can be drawn on paper. We will need to consider also *semantic* sketches, whose nodes and arrows are complex objects having some internal structure. For example, sketch Set whose nodes are all possible sets from some predefined universe and arrows are mappings between them, or sketch VarSet of variable sets and their mappings (see sect. B below), or sketch DB whose nodes are all possible states of some database and arrows are all possible transactions. Diagrams in such sketches are configurations of their items, which may have, or not have, some diagram properties.

In more detail, before modeling some class of constructs by sketches, we arrange the universe where these constructs live (are interpreted) as a graph U of certain objects and their mappings. Then we figure out a signature Σ of diagram predicates over U that we believe matter in our models. Finally, we convert the graph U into a semantic Σ-sketch U: for each label $P \in \Sigma$, the set of P-marked diagrams in this sketch, $D_{\mathsf{U}}(P)$, consists of all those diagrams in U of shape $Arity(P)$ that have the property P. Normally, sketches that we draw on a paper/computer display are specifications of finite fragments of the corresponding huge semantic sketch. An instance of such a specification-sketch S is nothing but a sketch mapping $I\colon\ \mathsf{S} \to \mathsf{U}$.

B Three formal semantic frameworks for OO visual modeling

B.1 Extensional view of object classes

Let C is a node, say, *Car*, in an object class diagram. In the pure extensional view on semantics of object classes, node *Car* should be interpreted as a set of *Car*-objects, $[\![\, Car \,]\!]$. However, as the system evolves, this set also evolves and so, extensionally, object classes are to be interpreted by *variable sets* (varsets).

B.1.1 Given a time moment t, we consider the collection of cars in question as a set $[\![\, Car \,]\!]^t$, and for another time moment, u, we will have another set, $[\![\, Car \,]\!]^u$. These sets are not mutually independent: they are inter-related by identifying different elements, say, $c' \in [\![\, Car \,]\!]^t$ and $c'' \in [\![\, Car \,]\!]^u$ as different appearances of the same car c. In this case we write $c' = c^t$ and $c'' = c^u$. So, if there are some verified reasons (a constructive proof as a mathematician would say) to consider c' and c'' to be the same object, this should be declared in an explicit form. The totality of all such declarations, irrespectively to ways how the information could be obtained, can be expressed by a family of binary relations

$$^t[\![\, SameCar \,]\!]^u \subseteq [\![\, Car \,]\!]^t \times [\![\, Car \,]\!]^u,\ t < u \text{ are time moments,}$$

so that $c' \in [\![Car]\!]^t$ and $c'' \in [\![Car]\!]^u$ have to be considered as different appearances of the same object iff $(c', c'') \in {}^t[\![SameCar]\!]^u$. We will call these relations *identity relations*.

B.1.2 Multiplicities of identity relations (*same*-associations) form a very important parameter of varsets; in a sense, they set different types of objects' ontology and must be explicitly specified. The most regular case is when multiplicities of all *same*-associations are 0..1, that is, the identity relations are relationships of one-one type. Then object identity for any pair $t \leq u$ is set by a pair of mutually inverse partially defined injections:

$$same^{t\to u}\colon [\![Car]\!]^t \to [\![Car]\!]^u \text{ and } same^{t\leftarrow u}\colon [\![Car]\!]^u \to [\![Car]\!]^t.$$

B.1.3 A special degenerated case of identity relations is when they are empty for all $t < u$. In such a case, object identities are defined for each given time moment (locally) but do not have a global lifetime meaning. A typical example of such objects is given by the familiar construct of pure association (relationship). More precisely, if we have an arrow span declared to be jointly monic for each single time moment (sect. A.2), and identity relation for its source are empty, then objects of the source can be considered as pure associations.

B.2 Behavioral view of object classes

Object classes are not only collections of object. They are also specifications of object behavior (technically, of attributes and methods) so that each object is a state transition system/state machine. To capture behavior into our formal model we proceed as follows.

B.2.1 States. We do not make any assumptions of what is a state of object: we merely introduce an abstract set $\langle\!\langle Car \rangle\!\rangle$ of possible states of *Car*-objects, a *state space*. Then at every moment t we have a totally defined mapping

(1) $\quad state^t\colon [\![Car]\!]^t \to \langle\!\langle Car \rangle\!\rangle$

which assigns to every *Car*-object existing at t its state. Note, while the mapping *state* and its domain are time dependent, the target – set $\langle\!\langle Car \rangle\!\rangle$ – does not depend on time and encompasses all the possible states of *Car*-objects.

Of course, the behavior of *Car*-object is not captured by just introducing the set $\langle\!\langle Car \rangle\!\rangle$ in our formal model. We may make observations about states without changing them, and we may change states as well. Hence, the set $\langle\!\langle Car \rangle\!\rangle$ is actually the carrier of a certain mathematical structure modeling observable behavior of *Car*-objects.

B.2.2 Transitions. A few mathematical structures for modeling behavior were proposed and elaborated: automata, labelled transition systems, abstract state machines, coalgebras and others. We will use coalgebra since it presents a compact and elegant framework for specifying behavior (see, e.g., [JR97]), and is naturally integrated with the varset part of our semantics.

Basics of the coalgebra approach for specifying behavior in our context are described in Appendix D. Briefly, signatures of all attributes and methods of the class *Car* jointly determine a functor $Car\mathbf{IFace}\colon \mathbf{Set} \to \mathbf{Set}$. Then their bodies are combined into a mapping

(2) $\quad \langle\!\langle car\mathsf{Meth} \rangle\!\rangle\colon \langle\!\langle Car \rangle\!\rangle \to \langle\!\langle Car \rangle\!\rangle.Car\mathbf{IFace}.$

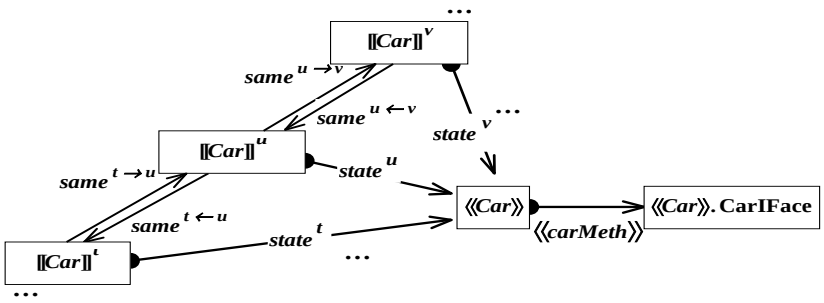


Figure 6: Behavioral semantics of object class, $t < u < v$ are time moments

This compact specification accomplishes our formal model of the behavioral side of semantics for the class *Car*.

B.2.3 Coalgebraic nature of *isPartOf*- and *isA*-constructs. UML documents describes composition in term of objects' deletions/creations: the composite (*whole*) manages deletion/insertions of its components (*part*s). However, this is only a fragment of common-sense intuition behind the *isPartOf* relationship between objects. Another, not less important, aspect of *isPartOf* is that properties of the *whole* are partly (not entirely!) determined by the *part*s' properties (see, e.g., [Kil99, WSW99]). In technical terms, this property determination means that the composite state machine inherits some of attributes and methods of the component state machines. Appendix D shows that this inheritance (not to be confused with *isA*-inheritance) can be precisely formalized as existence of a coalgebra morphism from (a certain reduct of) the composite coalgebra into (a similar reduct of) the product of component coalgebras. Particularly, it means that composition is a diagram predicate rather than conjunction of multiple *isPartOf*-predicates as it is assumed in UML. Aggregation is another diagram predicate, dual to composition in some precise sense (see [Dis02a] for details), also in contrast to its UML treatment as a conjunction of "weak" *isPartOf*-declarations for individual associations.

The behavioral aspects of *isA* relationship can be also formalized coalgebraically, [Pol00]. In this sense, in the formal semantics context, composition/aggregation and specialization/ generalization are essentially coalgebraic notions.

B.3 Extension and behavior together: Object classes as behavior sets

Syntactically, an object class, say, *Car*, is a specification of attributes of *Car*-objects and operations (methods) acting on them.

Its semantics assumes two components, extensional and behavioral. The former amounts to a variable set, that is, a sequence of sets indexed by time moments, $[\![\, Car \,]\!]^t, t \in T$. The behavioral component consists of a set $\langle\!\langle\, Car \,\rangle\!\rangle$ together with a mapping (2). These two components are connected together by mappings (1) sending objects to their states. The (infinite) diagram on Fig. 6 shows all these mappings together.

B.3.1 Definition. Let T be a partially ordered set (of time moments). A *(T-) behavioral set* C consists of the following components.

(ext) A variable set $[\![C]\!]$, that is, a family of sets $[\![C]\!]^t, t \in T$, with identity relations ${}^t[\![C]\!]^u$ for each pair $t \le u$.

(beh) A functor $\mathbf{IFace}_C\colon \mathbf{Set} \to \mathbf{Set}$, a set (of *states*) $\langle\!\langle C \rangle\!\rangle$ and a $\mathbf{IFace}_C$-coalgebra $\mathsf{Meth}_C\colon \langle\!\langle C \rangle\!\rangle \to \langle\!\langle C \rangle\!\rangle.\mathbf{IFace}_C$ over $\langle\!\langle C \rangle\!\rangle$.

(med) A family of mappings $state^t_C\colon [\![C]\!]^t \to \langle\!\langle C \rangle\!\rangle$, $t \in T$, assigning to C-objects their states at every moment t.

We can now summarize our semantics for object classes by saying that an object class is a behavioral set.

B.4 Plug-and-play semantics for sketches hidden in class diagrams.

A nice property of the behsets framework is that it is clearly modularized and consist of two components – extensional and behavioral. Correspondingly, semantics of modeling constructs used in the class diagrams is also "split" into the two components. Note also that our extensional component assumes the *variable* set semantics and hence is not static; particularly, objects' deletions/insertions are well modeled in the varset framework. One more modularization property is that the framework of variable sets can be naturally reduced to the framework of ordinary static sets: any varset $[\![C]\!]^t$, $t \in T$ gives rise to a static set $\bigcup_{t\in T} [\![C]\!]^t$.

So, the same sketch syntax can be interpreted in four different semantic frameworks (four Grothendieck's universes): $\mathsf{BehSet}, \mathsf{VarSet}, \mathsf{Set}$ and coAlg. These frameworks are related between themselves by reduction mappings, which go from the richer to the poorer universe and forget the "redundant" structure:

(3) $\mathsf{Set} \longleftarrow \mathsf{VarSet} \longleftarrow \mathsf{BehSet} \longrightarrow \mathsf{coAlg}.$

Then, as soon as we have an instance of sketch specification in a richer universe, we at once get the reduct of this instance in the poorer universe. It gives rise to a natural modularization mechanism for building formal semantics for OO modeling constructs. Owing to mappings (3), we may talk about a single yet multi-level semantic framework, where each level provides its own means for capturing the corresponding aspects of semantics, and the levels are mutually connected.

This idea is schematically presented in Table 1. On the static set level, we can model structural aspects of object class diagrams. Despite that it is a very poor universe for object class modeling, it allows us to clarify some important aspects still missed in UML, first of all, the diagrammatic (vs. string-based) logic nature of n-ary associations (Appendix C), or, say, a proper way of specifying qualified associations [Dis02b]. On the level of varset, more delicate aspects of semantics can be captured including object identity and identification, and objects' deletions/insertions as well; we might call this side of semantics *existential*. In the universe of coalgebras, state machines can be specified and manipulated, hence, objects' behavior can be modeled.[8] Finally, in the behset universe we can combine the extensional, including structural and existential, and the behavioral sides together to model quite complex constructs like *isPartOf*, *isA*, composition, aggregation, generalization and specialization (see [Dis02a] for details. And it seems that even semantics of simple binary associations can require the full power of the behset framework – see a discussion in [Ste01]).

<table>
<tr><td rowspan="3">Formal framework</td><td colspan="3">Behavioral sets</td></tr>
<tr><td colspan="2">Variable sets</td><td rowspan="2">Coalgebra</td></tr>
<tr><td>Static sets</td><td>Changeability in time</td></tr>
<tr><td>For modeling:</td><td>Structural aspects of class diagrams</td><td>Existential aspects of class diagrams: Objects' identity and deletions/insertions</td><td>Objects' behavior</td></tr>
<tr><td>Examples:</td><td>N-ary associations, qualifications, hierarchies of composition, aggregation and generalization associations</td><td>Frozen references, existential aspects of isA- and isPartOf</td><td>Statechart diagrs., activity diagrs.</td></tr>
<tr><td rowspan="2">Integrated in:</td><td colspan="2">Extensional side of ontology of modeling constructs (composition, aggregation, specialization, generalization)</td><td>Behavioral side of objects' ontology</td></tr>
<tr><td colspan="3">Complete (?) semantics for OO visual modeling</td></tr>
</table>

Table 1: Integrated formal semantics for OO visual modeling

C Multiplicities for n-ary associations: *Much ado about nothing.*

The problem appeared on the stage right after Peter Chen entered ER-diagrams into the world of visual modeling, and since then until now is still discussed (sometimes in a rather heated way) in the literature of both theoretical and industrial orientations. The most complete survey can be found in [GLn01], which refers to more than ten recent papers specially devoted to the problem and to eight basic monographs on conceptual modelling and OOA&D touching the problem. The authors of [GLn01] carefully analyze the existing approaches and on this base try to solve the problem. However, as they themselves state in the conclusion, the problem still remains unsolved and present an essential flaw in the UML 1.3.

There is nothing surprising in that failure. As soon as one has precise sketch specification described below, the entire problem becomes at once understandable as a typical case of known phenomenon: a pseudo-mathematical game with ill-formed terms. In other words, the problem of multiplicities for N-ary associations is actually a pseudo-problem caused by using an unsuitable specification language rather than the subject matter as such.

C.1 The most general case of N-ary association is specified by a sketch S_{1A} in the 1st column of Table 2, where the marker [1-1] declares the family of arrows to be jointly monic and, hence, objects of set A can be considered as tuples (sect. A.2). This simple picture shows that multiplicity properties of an association are nothing but multiplicity properties of its projection mappings. The latter are ordinary mappings for which multiplicities have their ordinary meaning, and multiplicities of the i-th projection are not anyhow related to multiplicities of the j-th. In addition, projection mappings are always totally defined and single-valued, and hence the only qualitatively important property that can be declared for a projection arrow is whether it is covering (see sect A.1). So, it is a must to specify covering properties of projection arrows in visual models of N-ary associations. However, with the UML's way of specifying N-ary associations, one cannot express such properties: the same UML diagram D_1 in

Table 2 corresponds to different sketches S_{1A} and S_{1B}.

C.2 Another aspect of N-ary associations which is also included into their multiplicity properties actually has quite another nature. The point is that for a given N-ary association, that is, jointly monic N-span $(p_1, \dots, p_N)$, some combination of its projections, say, $p_{i_1}, \dots, p_{i_k}$, $1 \le i_1 \le i_k \le N$, can be also one-one (in the relational data model terminology, the entire N-tuple of projections is then a superkey). For example, the actual meaning of UML-diagram D_2 in Table 2 is specified by sketch S_2 (compare it with sketch S_{1A}). However, the conventional approach (inherited by UML) to this type of constraints fails in a little bit more complicated situations. Consider, for example, sketches S_{3A} and S_{3B}: they specify essentially different associations (S_{3B} entails S_{3A} but not the converse), whose UML's representations coincide (diagram D_3). In other words, the UML's visualization cannot separate two different specifications.

C.3 One might argue that our sketch specifications of associations do not distinguish between pure associations (relationships) and association classes. It is indeed so until we consider the issue within the framework of static sets, but it can be well managed in the framework of variable sets (sect. B.1). For the latter, we must carefully distinguish between the two related yet essentially different ways of identifying objects by arrow spans: *local* identification for each given time moment (sect. B.1.3) and *global* identification valid throughout the entire life time of the system. Correspondingly, we have two related yet different diagram predicates: *locally* and, respectively, *globally jointly monic* spans, details can be found in [DK03, Dis02a].

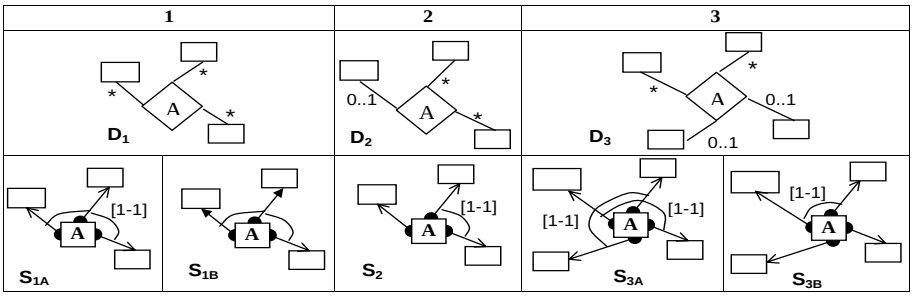


Table 2: Multiplicity constraints for N-ary associations

D Coalgebraic formal semantics for objects' behavior and relationships

All mappings considered in this section are totally defined and single-valued.

D.1 Objects as coalgebras.

Let *Car* be an object class and $\langle\!\langle\, Car \,\rangle\!\rangle$ is its state space, that is, a set of states that *Car*-objects can have during their lifetime (see B.2.1).

D.1.1 We can make observations of states, which are modeled by mappings of state spaces into value domains. For example, an attribute *fuelAmount* is a mapping $\langle\!\langle\, fuelAmount \,\rangle\!\rangle : \langle\!\langle\, Car \,\rangle\!\rangle \to \texttt{Float(0..100)}$ that shows the percentage of how full

is the tank in a given state. We can also change states by applying to them predefined operations (methods, in the OO jargon), for example, *fillTank(p)*, where p is a `Float(0..100)`-parameter pointing to what extent (in percents) the tank must be filled in. Such a method is modeled by a mapping $\langle\!\langle\, \mathit{fillTank}(p) \,\rangle\!\rangle \colon \langle\!\langle\, \mathit{Car} \,\rangle\!\rangle \to \langle\!\langle\, \mathit{Car} \,\rangle\!\rangle$ that sends any state s to a state s' with $s'.\langle\!\langle\, \mathit{fuelAmount} \,\rangle\!\rangle = p$ if $s.\langle\!\langle\, \mathit{fuelAmount} \,\rangle\!\rangle < p$ and $s' = s$ otherwise.

A class normally has many attributes and methods, and compact specification of the corresponding state machine might be a problem. However, coalgebra manages the issue in a very elegant and really compact way ([JR97]).

D.1.2 Coalgebraically, a method or attribute m acting on a state space S is modelled by a mapping

(4) $\quad \langle\!\langle\, m \,\rangle\!\rangle \colon S \to S.m\mathbf{IFace}$

where $m\mathbf{IFace} \colon \mathbf{Set} \to \mathbf{Set}$ is a construction defined on sets, which builds from any set X a certain set $m\mathbf{IFace}(X)$ according to m's interface. In category theory such mappings are called functors. For instance, in our *Car*-example, $S = \langle\!\langle\, \mathit{Car} \,\rangle\!\rangle$ and for the attribute m= *fuelAmount*, we set $X.m\mathbf{IFace} =$ `Float(0..100)` independently of set X. Then the body of this attribute is a mapping of type (4) as required in coalgebra framework.

To present the method $m = \mathit{fillTank}(p)$ in this way, we first note that having a mapping $f \colon P \times S \to S$ (with P a set of parameters) is equivalent to having a mapping $f^{\#} \colon S \to \mathbf{Map}(P, S)$, where $\mathbf{Map}(P, S)$ denotes the set of all mappings from P to S. Now, if for *fillTank***IFace** we take the functor $\mathbf{Map}(P, \_) \colon \mathbf{Set} \to \mathbf{Set}$ that assigns to any set X the set $\mathbf{Map}(P, X)$ of all mappings from P to X, $P =$ `Float(0..100)`, then the body of the method can be equivalently presented by the mapping

$$\langle\!\langle\, \mathit{fillTank} \,\rangle\!\rangle \colon \langle\!\langle\, \mathit{Car} \,\rangle\!\rangle \to \langle\!\langle\, \mathit{Car} \,\rangle\!\rangle.\mathbf{Map}(P, \_)$$

of type (4) as required in coalgebra.

D.1.3 If we have a collection of methods/attributes $\mathsf{Meth} = \{m_i \mid i = 1, ..., n\}$ acting on the same state space S, we collect their interfaces $m_i\mathbf{IFace}$'s into a single interface $\mathbf{IFace} = \prod_{i=1,...,n} m_i\mathbf{IFace} \colon \mathbf{Set} \to \mathbf{Set}$ by defining for any set X,

$$X.\mathbf{IFace} = X.m_1\mathbf{IFace} \times \ldots \times X.m_n\mathbf{IFace}.$$

Then we can collect the bodies of methods $\langle\!\langle\, m_i \,\rangle\!\rangle$'s into a single mapping

$$\langle\!\langle\, \mathit{meth} \,\rangle\!\rangle \colon S \to S.\mathbf{IFace} \text{ by defining } s.\langle\!\langle\, \mathit{Meth} \,\rangle\!\rangle \stackrel{\text{def}}{=} (s.\langle\!\langle\, m_1 \,\rangle\!\rangle, ..., s.\langle\!\langle\, m_n \,\rangle\!\rangle)$$

for any state $s \in S$. In this way a complex state machine with multiple methods can be specified in a unified compact way.

D.1.4 Definition: State machines as coalgebras. In the coalgebra framework a state machine consists of the following components:

(i) a functor $\mathbf{IFace} \colon \mathbf{Set} \to \mathbf{Set}$ specifying the signatures of the methods jointly,

(ii) a state space $S \in \mathbf{Set}$,

(iii) a mapping $\langle\!\langle\, \mathsf{Meth} \,\rangle\!\rangle \colon S \to S.\mathbf{IFace}$ specifying the bodies of the methods jointly.

A triple as above is called a **IFace**-coalgebra with the carrier set S; functor **IFace** is then called the *signature* of the coalgebra. There may be many different **IFace**-coalgebras over the same carrier set, and of course **IFace**-coalgebras can be defined over different carrier sets.

D.2 Coalgebraic model of *isPartOf*

For a generic example, let's consider the statement "engine is a part of car". Its semantics has the two sides: extensional and behavioral, mutually related as described in B.2. In the present section we consider the behavioral side, and treat it coalgebraically. It means that we enter onto the stage two coalgebras (state machines):

$$\langle\!\langle\, car\mathsf{Meth} \,\rangle\!\rangle\colon \langle\!\langle\, Car \,\rangle\!\rangle \to \langle\!\langle\, Car \,\rangle\!\rangle.Car\mathbf{IFace} \text{ and } \langle\!\langle\, eng\mathsf{Meth} \,\rangle\!\rangle\colon \langle\!\langle\, Eng \,\rangle\!\rangle \to \langle\!\langle\, Eng \,\rangle\!\rangle.Eng\mathbf{IFace},$$

where $\langle\!\langle\, Xxx \,\rangle\!\rangle$ denotes the state space of class Xxx, $Xxx\mathsf{Meth}$ is its set of methods, whose signatures jointly give rise to a functor $Xxx\mathbf{IFace}\colon \mathbf{Set} \to \mathbf{Set}$, the product of individual method interfaces; the mapping $\langle\!\langle\, Xxx\mathsf{Meth} \,\rangle\!\rangle$ is the product of individual method bodies.

Evidently, declaring an engine as a part of car means that these two coalgebras are somehow related. Our goal is to figure out this relation.

D.2.1 First of all, since engines are parts of cars, states of cars should "include" (in one or another sense) states of their engines. Formally, it can be modeled by a mapping $\langle\!\langle\, has \,\rangle\!\rangle\colon \langle\!\langle\, Car \,\rangle\!\rangle \to \langle\!\langle\, Engine \,\rangle\!\rangle$ that assigns to a car state its *Engine*-component.

Moreover, methods of the class *Engine* (at least, some of them) should give rise to the corresponding methods of the class *Car*. For example, an *Engine*-method of replacing sparking plugs, $\langle\!\langle\, rePlug \,\rangle\!\rangle^{Eng}\colon \langle\!\langle\, Engine \,\rangle\!\rangle \to \langle\!\langle\, Engine \,\rangle\!\rangle$, gives rise to the corresponding *Car*-method so that normally we say "I'm going to replace sparking plugs in my car" rather than "I'm going to replace plugs in the engine of my car". It means that there is a *Car*-method $\langle\!\langle\, rePlug \,\rangle\!\rangle^{Car}\colon \langle\!\langle\, Car \,\rangle\!\rangle \to \langle\!\langle\, Car \,\rangle\!\rangle$, related to $\langle\!\langle\, rePlug \,\rangle\!\rangle^{Engine}$ in the following way: for any *Car*-state $s \in \langle\!\langle\, Car \,\rangle\!\rangle$, the following equation holds:

$$(5)\qquad s.\langle\!\langle\, rePlug \,\rangle\!\rangle^{Car}.\langle\!\langle\, has \,\rangle\!\rangle = s.\langle\!\langle\, has \,\rangle\!\rangle.\langle\!\langle\, rePlug \,\rangle\!\rangle^{Eng}.$$

In words: *Engine*-projection of a transformed *Car*-state is the transformed *Engine*-projection of an initial *Car*-state.

The method we have just considered has a very simple interface, without parameters, considerations of cases and the like, just direct transformation of a state into a state. However, coalgebra allows us to generalize considerations above for any methods with arbitrarily complex interfaces. All the complexity is hidden in the interface functor while the general schema remains the same. Namely, let m be an *Engine*-method,

$$\langle\!\langle\, m \,\rangle\!\rangle^{Engine}\colon \langle\!\langle\, Engine \,\rangle\!\rangle \to \langle\!\langle\, Engine \,\rangle\!\rangle.m\mathbf{IFace},$$

with interface $m\mathbf{IFace}\colon \mathbf{Set} \to \mathbf{Set}$. Suppose that it acts also on states of engines-as-parts-of-cars, then we have also a *Car*-method

$$(6)\qquad \langle\!\langle\, m \,\rangle\!\rangle^{Car}\colon \langle\!\langle\, Car \,\rangle\!\rangle \to \langle\!\langle\, Car \,\rangle\!\rangle.m\mathbf{IFace}.$$

Finally, if the method acts on an engine-as-part-of-car in exactly the same way as it acts on an isolated engine, the following diagram holds commutative:

$$(7)\qquad \begin{array}{ccc} \langle\!\langle\, Car \,\rangle\!\rangle & \xrightarrow{\quad\langle\!\langle\, m \,\rangle\!\rangle^{Car}\quad} & \langle\!\langle\, Car \,\rangle\!\rangle.m\mathbf{IFace} \\ \Big\downarrow{\scriptstyle \langle\!\langle\, has \,\rangle\!\rangle} & & \Big\downarrow{\scriptstyle \langle\!\langle\, has \,\rangle\!\rangle.m\mathbf{IFace}} \\ \langle\!\langle\, Engine \,\rangle\!\rangle & \xrightarrow{\quad\langle\!\langle\, m \,\rangle\!\rangle^{Engine}\quad} & \langle\!\langle\, Engine \,\rangle\!\rangle.m\mathbf{IFace} \end{array}$$

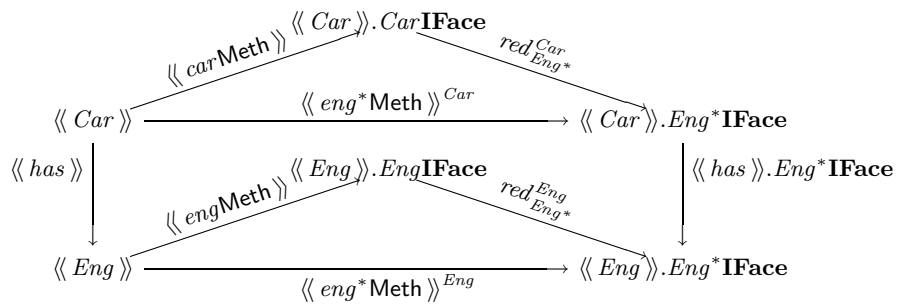


Figure 7: *Engine* is a part of *Car*: behavioral aspects. $eng^*\mathsf{Meth}$ denotes the set of *Engine*-methods which act on engine-as-part-of-car in exactly the same way as they act on isolated engines; Eng^***IFace** is their joint interface.

In such a case we will also say that *Car*-objects **-inherit* the method m; the index $*$ is introduced here to distinguish this inheritance from the familiar inheritance related to *isA*-relationship.

D.2.2 The engine in a car works in a special environment that can essentially change the bodies and results of some of *Engine*-methods (in comparison with their behavior in isolated engines) or block them at all. So, we may not assume that every *Engine*-method is *-inherited in *Car*. Yet we assume that there is a non-empty set

$$Eng^*\mathsf{Meth} = \{m_j \mid j = i_1, \ldots, i_k\} \subset Eng\mathsf{Meth} = \{m_i \mid i = 1, \ldots, n\}$$

of *-inherited *Engine*-methods, *ie*, those for which diagram (7) is commutative. We can combine interfaces of these methods into an interface

$$Eng^*\mathbf{IFace} = \prod_{j=i_1,\ldots,i_k} m_j\mathbf{IFace}\colon\ \mathbf{Set} \to \mathbf{Set},$$

like it was shown in D.1.3, and then there is an evident reduction mapping

$$red^{Eng}_{Eng^*}\colon\ \langle\!\langle\, Engine \,\rangle\!\rangle.Eng\mathbf{IFace} \to \langle\!\langle\, Engine \,\rangle\!\rangle.Eng^*\mathbf{IFace}$$

that forgets the components not occurring into Eng^***IFace**, and the lower triangle diagram on Fig. 7 is commutative.

Since each method m from $Eng^*\mathsf{Meth}$ is *-inherited, the entire $Eng^*\mathsf{Meth}$-subinterface of class *Engine* is embedded into the *Car*-interface,

$$h\colon\ Eng^*\mathsf{Meth} \hookrightarrow Car\mathsf{Meth}$$

Hence, since interface functors are products indexed by method names, see sect. D.1.3, there is also a reduction mapping

$$red^{Car}_{Eng^*}\colon\ \langle\!\langle\, Car \,\rangle\!\rangle.Car\mathbf{IFace} \to \langle\!\langle\, Car \,\rangle\!\rangle.Eng^*\mathbf{IFace}$$

such that the upper triangle subdiagram on Fig. 7 is commutative.

Commutativity of the upper triangle says that $Eng^*\mathsf{Meth}$-methods are simply considered as *Car*-methods, this condition concerns only the method signatures but says nothing about *how* Eng^*-methods work on *Car*-states. To express formally that

these methods act on an engine-as-part-of-car exactly like they act on an isolated engine, we require also commutativity of the rectangle subdiagram on Fig. 7.

Thus, the entire diagram on Fig. 7 is commutative, and this precisely captures the intuition of the statement that "*Car*-objects *-inherit some of *Engine*-object methods". On the other hand, commutativity of the rectangle means that mapping $\langle\!\langle\, has \,\rangle\!\rangle$ is a coalgebra morphism. So, a quite simple coalgebraic statement formalizes a non-trivial intuitive concept.

D.2.3 Some properties of *-inheritance. Let objects of class C are declared as parts of objects of class B. Behaviorally, it means that there is an embedding

$$(8) \qquad h_{CB} \colon {}_{C^*}\mathsf{Meth} \hookrightarrow {}_{B}\mathsf{Meth}$$

for some subset ${}_{C^*}\mathsf{Meth} \subset {}_{C}\mathsf{Meth}$ of C-methods, which are *-inherited by B-objects.

There are two extreme cases: (i) when ${}_{C^*}\mathsf{Meth}$ is empty, and (ii) when ${}_{C^*}\mathsf{Meth} = {}_{C}\mathsf{Meth}$. In the case (i), the parts are either lose their individuality and "dissolved" in the whole (there are no part's methods in the whole interface), or the behavior of parts is entirely changed within B-objects (all the part's methods are overridden) and instead of C-objects, we have inside of B-objects something different. In the case (ii), the parts keep their behavior within the whole entirely unchanged, they are mechanically included into the whole and not affected by the whole environment.

Note also that *-inheritance is not transitive. Let C-objects are declared to be parts of B-objects, while the latter are declared to be parts of A-objects. Then we have two embedding mappings, (8) above and

$$(9) \qquad h_{BA} \colon {}_{B^*}\mathsf{Meth} \hookrightarrow {}_{A}\mathsf{Meth}$$

for some subset ${}_{B^*}\mathsf{Meth} \subset {}_{B}\mathsf{Meth}$ of B-methods. Now, let ${}_{C^{**}}\mathsf{Meth}$ denotes the set of those C-methods from ${}_{C^*}\mathsf{Meth}$ that get into ${}_{B^*}\mathsf{Meth}$ with the mapping (8). Methods from ${}_{C^{**}}\mathsf{Meth}$ are *-inherited by A-objects through their *-inheritance by B-objects, and this would provide transitivity, but generally speaking the set ${}_{C^{**}}\mathsf{Meth}$ can be empty for non-empty sets ${}_{C^*}\mathsf{Meth}$ and ${}_{B^*}\mathsf{Meth}$! For example, consider a small spring in an engine. For an engine, this spring is sufficiently important and (some of) its methods occur into the *Engine*-interface (*-inherited). However, when we consider engines as parts of cars, our spring becomes of too little importance and its methods does not occur into the *Car*'s interface. (Or, consider an opposite hypothetical situation that the *Car*-environment so essentially changes functioning of engines and their parts, that springs behave themselves in engines-as-parts-of-cars quite differently from their behavior in engines). Thus, in general, *-inheritance is not transitive.

This leads us to a counterintuitive conclusion that *isPartOf*-relationship is not transitive. Note, however, that *isPartOf* encompasses extensional aspects (physical containment, including propagation of deletion/insertion of objects, see [DK03, Dis02a]), and behavioral aspects (modelled by *-inheritance). Of course, the extensional side of *isPartOf* must be transitive, while the behavioral, as we have just seen, is not. Then, it is reasonable to consider two relationships between classes: *extensional isPartOf* and *behavioral isPartOf*. They are closely related but do not coincide: the first one often implies the second but not always – recall our extremal case (i) when nothing from the parts is *-inherited. On the other hand, we can well imagine behavioral *isPartOf* without physical containment, consider, for example, something like a federation of social units.

D.3 Coalgebraic model of composition

The construct of composition assumes that a few *isPartOf*-relationships are declared for the same *whole*, for example, class *Car* is declared to be a composition of classes *Body*, *Chassis*, *Engine*.

D.3.1 State spaces. It means, first of all, that we have three mappings,

$$\langle\!\langle\, has \,\rangle\!\rangle_i \colon \langle\!\langle\, Car \,\rangle\!\rangle \to \langle\!\langle\, Part_i \,\rangle\!\rangle, i = 1, 2, 3,$$

for, respectively, $Part_{1,2,3}$=*Body*,*Chassis*,*Engine*. Hence, there is a mapping

$$\langle\!\langle\, has \,\rangle\!\rangle \colon \langle\!\langle\, Car \,\rangle\!\rangle \to \langle\!\langle\, Prod \,\rangle\!\rangle, \text{ with } \langle\!\langle\, Prod \,\rangle\!\rangle \stackrel{\text{def}}{=} \prod_{i=1,2,3} \langle\!\langle\, Part_i \,\rangle\!\rangle,$$

which assigns to each state of a *Car*-object C a triple of states of C's components.

In general, the mapping $\langle\!\langle\, has \,\rangle\!\rangle$ is neither covering nor one-one. It is not covering since there are well possible triples of component states, which cannot appear as a *Car*-state due to some compatibility conditions (*negative* emergency). The mapping is not one-one since there are well possible different *Car*-states with the same triples of component states, for example, if the car's states include financial parameters, say, the car's price and the like (*positive* emergency). These qualities of mapping $\langle\!\langle\, has \,\rangle\!\rangle$ capture some aspects of the well known phenomenon: composition is usually something richer than just sum of its parts.

D.3.2 Methods. Each component has its own set of methods $Meth_i$, with joint interface $\mathbf{IFace}_i$, and each component is a coalgebra (state machine) in its own signature:

$$(10) \quad \langle\!\langle\, part_i\mathsf{Meth} \,\rangle\!\rangle \colon \langle\!\langle\, Part_i \,\rangle\!\rangle \to \langle\!\langle\, Part_i \,\rangle\!\rangle.Part_i\mathbf{IFace}, i = 1, 2, 3.$$

We will also use synonymic denotations: Meth_i for $Part_i\mathsf{Meth}$ and $\mathbf{IFace}_i$ for $Part_i\mathbf{IFace}$.

Several state machines can be composed/combined in a variety of ways. Syntax for encoding such combinations with substantial yet informal semantics have been studied in various theories of processes (process calculi), see, e.g., [Mil99]. An adequate coalgebraic treatment (and a formal semantics at all) of process calculi is still an open problem, which appears to be extremely intricate both technically and conceptually, see, e.g., an attempt in [Wol99]. Our goal in this paper is very modest w.r.t. this problem. We will consider a very simple type of combining state machines with an immediate (but still not too simple technically) coalgebraic treatment. Namely, having coalgebras (10), we define their *product coalgebra*,

$$(11) \quad \langle\!\langle\, prod\mathsf{Meth} \,\rangle\!\rangle \colon \langle\!\langle\, Prod \,\rangle\!\rangle \longrightarrow \langle\!\langle\, Prod \,\rangle\!\rangle.Prod\mathbf{IFace}$$

in the following way. The state space is the product of state spaces $\langle\!\langle\, Prod \,\rangle\!\rangle = \prod_{i=1,2,3} \langle\!\langle\, Part_i \,\rangle\!\rangle$. Methods are triples (m_1, m_2, m_3) with $m_i \in Meth_i$, $i = 1, 2, 3$, which act on the product state space $\langle\!\langle\, Prod \,\rangle\!\rangle$ in a natural way: they send a state $s = (s_1, s_2, s_3)$ to a triple

$$t = (s_1.\langle\!\langle\, m_1 \,\rangle\!\rangle^{Part_1}, s_2.\langle\!\langle\, m_2 \,\rangle\!\rangle^{Part_2}, s_3.\langle\!\langle\, m_3 \,\rangle\!\rangle^{Part_3}) \in \prod_{i=1,2,3} \langle\!\langle\, Part_i \,\rangle\!\rangle.Part_i\mathbf{IFace}.$$

Note, t is not an element of $\langle\!\langle\, Prod \,\rangle\!\rangle.\prod_{i=1,2,3} \mathbf{IFace}_i$ and hence the type of mapping $\langle\!\langle\, (m_1, m_2, m_3) \,\rangle\!\rangle$ does not conform to the $(\prod_i \mathbf{IFace}_i)$-coalgebra pattern. However, if the component interface functors have some nice technical properties, then we may

hope that they can be joined in some natural way into a functor *Prod***IFace** such that the triple above would belong to $\langle\!\langle\, Prod \,\rangle\!\rangle.Prod\mathbf{IFace}$. Details still need to be checked.

D.3.3 Product and components. Methods of each component are embedded into the methods of the product. For example, a method $m \in Meth_2$ of the second component gives rise to a product method (id_1, m, id_3), where id_i is the identity mapping of $\langle\!\langle\, Part_i \,\rangle\!\rangle$. Then we have embeddings, $h_i\colon\ Part_i\mathsf{Meth} \hookrightarrow Prod\mathsf{Meth}$, and correspondingly, signature reductions, $red^{Prod}_{Part_i}\colon\ Prod\mathbf{IFace} \to Part_i\mathbf{IFace}$, $i = 1, 2, 3$, such that the diagram (12) is commutative for every $i = 1, 2, 3$.

$$
\begin{array}{ccc}
\langle\!\langle\, Prod \,\rangle\!\rangle & \xrightarrow{\ prod\mathsf{Meth}\ } & \langle\!\langle\, Prod \,\rangle\!\rangle.Prod\mathbf{IFace} \\
\Vert & & \downarrow red^{Prod}_{Part_i} \\
\langle\!\langle\, Prod \,\rangle\!\rangle & \xrightarrow{\ \langle\!\langle\, part_i\mathsf{Meth} \,\rangle\!\rangle^{Prod}\ } & \langle\!\langle\, Prod \,\rangle\!\rangle.Part_i\mathbf{IFace} \\
proj_i \downarrow & & \downarrow proj_i.Part_i\mathbf{IFace} \\
\langle\!\langle\, Part_i \,\rangle\!\rangle & \xrightarrow{\ \langle\!\langle\, part_i\mathsf{Meth} \,\rangle\!\rangle^{Part_i}\ } & \langle\!\langle\, Part_i \,\rangle\!\rangle.Part_i\mathbf{IFace}
\end{array}
\tag{12}
$$

In that diagram, the "arrow" $\Vert\Vert$ denotes the identity mapping and introduced only for convenience of drawing the diagram, the upper rectangle (in fact, triangle) just shows the reduction of signatures and the lower rectangle means that mapping $proj_i$ is a coalgebra morphism (that is, methods of *i-part* act on the i-component of the product exactly like they act on an isolated *i-part*).

D.3.4 Whole and product. What is the relationship of the *Car*-coalgebra (state machine) to this product coalgebra? Not every possible product method (generated by triples (m_1, m_2, m_3)) is *-inherited by *Car*-objects, and we should repeat out consideration in sect. D.2.2 with the product coalgebra instead of *Engine*-coalgebra.

Let $Prod^*\mathsf{Meth}$ denote the set of product methods that act on *Car*-states without overriding (*-inherited), and $Prod^*\mathbf{IFace}$ is their joint interface. Let ${Part_i}^*\mathsf{Meth}$ is the set of those $Part_i$-methods that get into $Prod^*\mathsf{Meth}$ by the mapping h_i (sect.D.3.3

$$
{Part_i}^*\mathsf{Meth} \stackrel{\text{def}}{=} \{m \in Part_i\mathsf{Meth} \mid m.h_i \in Prod^*\mathsf{Meth}\}\,. \tag{13}
$$

We require this set to be non-empty.

Now we have a commutative diagram of embeddings of sets of methods:

$$
\begin{array}{ccccc}
Part_i\mathsf{Meth} & & \xrightarrow{\ h_i\ } & & Prod\mathsf{Meth} \\
v_i^* \uparrow & & [\mathtt{coIm}] & & v_P \uparrow \\
{Part_i}^*\mathsf{Meth} & \xrightarrow{\ h_i^*\ } & Prod^*\mathsf{Meth} & \xrightarrow{\ h_{P^*}\ } & Whole\mathsf{Meth},
\end{array}
\tag{14}
$$

where the marker [coIm] states that the node ${Part_i}^*\mathsf{Meth}$ with the arrows v_i^* and h_i^* are produced from the node $Prod\mathsf{Meth}$ with the two incoming arrows by applying to them the *diagram operation* of taking coimage (so that actually the diagram is a sketch!). Diagram (14) gives rise to a corresponding commutative diagram of reductions of interface functors:

$$
\begin{array}{ccccc}
Part_i\mathbf{IFace} & & \xleftarrow{\ red^{Prod}_{Part_i}\ } & & Prod\mathbf{IFace} \\
red^{Part_i}_{Part_i^*} \downarrow & & & & \downarrow red^{Prod}_{Prod^*} \\
{Part_i}^*\mathbf{IFace} & \xleftarrow{\ red^{Prod^*}_{Part_i^*}\ } & Prod^*\mathbf{IFace} & \xleftarrow{\ red^{Whole}_{Prod}\ } & Whole\mathbf{IFace}
\end{array}
\tag{15}
$$

Finally, using these reductions for matching the types of the mappings involved we can join the diagram on Fig. 7 written in terms of *Whole* for *Car* and *Product* instead of *Engine*, with the diagrams (12) for $i = 1, 2, 3$: the result is presented by commutative diagram on Fig. 8.

Commutativity of the three triangle sub-diagrams and the middle rectangle (in fact, also triangle since $\equiv$-arrow just "splits" the same node) fulfills a technical (yet important) role of matching the signatures of coalgebras. Commutativity of the upper rectangle states that methods from the $Prod^*\mathsf{Meth}$-subset of the product coalgebra methods act on the whole without overriding, *-inherited by the whole state machine from the product state machine (that, in its turn, is the coalgebraic product of the component state machines). Commutativity of the down rectangle states that methods from the $Part_i{}^*\mathsf{Meth}$-subset of the i-th component methods act on the product without overriding, that is, *-inherited by the product state machine from its i-th component state machine. Since the sub-rectangles are commutative, the outer big rectangle is commutative too, and it means that some of the methods of every component are *-inherited by the composite. That is, by our definition of formal *isPartOf*, every component of the composite is its part in the formal sense too (as it should be). The converse implication is not so simple, see the next section.

To summarize in words, declaring objects of class *Whole* to be compositions of parts $Part_i$, $i = 1, 2, 3$, means, behaviorally, that there is a coalgebra homomorphisms from some reduct of the *Whole*'s state machine into a similar reduct of the coalgebraic product of the component state machines.

E Model management via arrows

The goal of this section is to outline a formal arrow-based framework for model management (MMt). The framework is entirely abstract in the sense that we will consider MMt constructs without making any assumptions about what a model is. Hence, our definitions of queries, views, model integration and other constructs will be applicable to models of arbitrary kind (UML diagrams of a specific type, ER-diagrams, sketches, relational database schemas *etc*) – just what the theory of MMt should provide.

E.1 Getting started. We will assume that along with models there are given model morphism (mappings between models), and then we will define all constructs of MMt in terms of models and morphisms. Reformulation of the constructs of interest in terms of mappings is the essence of *arrow thinking* underlying CT. It can be also phrased in OO-terms as that objects (models in our case) show themselves only through arrow interfaces.

Thus, we suppose that we are given a (directed multi)graph $\mathcal{R}$, whose nodes are called *models* and arrows are *model morphisms (mappings)*. This graph can be considered as a huge repository of models/specifications of a given type. For example, the models can be ER-diagrams, and their morphisms are mappings that send E-nodes to E-nodes, R-nodes to R-nodes and attributes to attributes so that their incidence is preserved. Another example is when the models are sketches and morphisms are sketch mappings, *ie*, mappings of the carrier graphs compatible with diagram markers, sect. A.4. Or else models can be relational database schemas, and their morphisms are mappings that send relation names to relation names of the same attribute schema in a way compatible with dependencies.

So, our goal is to elaborate MMt constructs in terms of $\mathcal{R}$. A series of samples below shows that it can be really done.

E.2 Instances as mappings. Let $Nds\mathcal{R}$ denotes the class of $\mathcal{R}$-nodes. We suppose that there is a special subclass $\mathcal{U} \subset Nds\mathcal{R}$ of models called *semantic universes*. For example, if models are ER-diagrams, a *ER-universe* is a huge "ER-diagram", ER, whose E-nodes are all possible finite sets of entities (from some predefined universe) and R-nodes are all possible finite relations over them. Actually, we need to define ER-model mappings and ER-universes in such a way that a valid instance of ER-diagram D can be considered as a mapping $I\colon \mathsf{D} \to \mathsf{ER}$ and conversely. If models are sketches (in a given signature Σ), then a semantic universe is a semantic sketch as discussed in sect. A.5. Typical examples are universes of sets, or varsets, or behsets (sect. B) and their mappings, which are arranged as sketches $\mathsf{Set}, \mathsf{VarSet}, \mathsf{BehSet}$ respectively.

These examples are generic and the same idea can be applied for many types of specifications. Thus, having a repository $\mathcal{R}$ with a class of semantic universes $\mathcal{U} \subset Nds\mathcal{R}$, we define an *instance* of model S to be a mapping $I\colon \mathsf{S} \to \mathsf{U}$ for some $\mathsf{U} \in \mathcal{U}$. Then the set of instances of a model S in a given universe $\mathbf{U}$, $Inst^{\mathbf{U}}(\mathsf{S})$, appears as the set of mappings $I\colon \mathsf{S} \to \mathsf{U}$, where U is $\mathbf{U}$ considered as a (huge) semantic model, that is, a node $\mathsf{U} \in \mathcal{U}$.

A major construct of MMt (still without a common name) is that any model mapping $f\colon \mathsf{S}_1 \to \mathsf{S}_2$ gives rise to a mapping of instances $f^*\colon Inst^{\mathbf{U}}(\mathsf{S}_2) \to Inst^{\mathbf{U}}(\mathsf{S}_1)$, note the reverse of direction. The instances-as-mappings view readily explains it: given f, any instance I of S_2 gives rise to an instance $f^*(I)$ of S_1 by composition of mappings, $f^*(I) \stackrel{\text{def}}{=} f \mathbin{⩥} I\colon \mathsf{S}_1 \to \mathsf{S}_2 \to \mathsf{U}$.

E.3 Submodels. The submodel relationship when model S_1 is a subset of model S_2, or model S_2 extends/inherits model S_1, can be considered as a particular case of model mapping when the latter is inclusion, $i\colon \mathsf{S}_1 \hookrightarrow \mathsf{S}_2$. If $I\colon \mathsf{S}_2 \to \mathsf{U}$ is an instance of S_2, then $i^*(I) = i \mathbin{⩥} I$ is nothing but the restriction of I on S_1. In other words, $i^*\colon Inst(\mathsf{S}_2) \to Inst(\mathsf{S}_1)$ is a sort of a reduction mapping that makes nothing but forgets the extra structure of S_2-instances and thus makes them S_1-instances.

E.4 Switching between semantic universes. Often we have more than one intended semantic universe for interpreting specifications. Normally, these universes can be related by a morphism, say, $r\colon \mathsf{U}_1 \to \mathsf{U}_2$. It gives rise to a mapping of instances $r_*\colon Inst^{\mathbf{U}_1}(\mathsf{S}) \to Inst^{\mathbf{U}_2}(\mathsf{S})$ defined by composition again: for $I \in Inst^{\mathbf{U}_1}(\mathsf{S})$, $r_*(I) \stackrel{\text{def}}{=} I \mathbin{⩥} r : \mathsf{S} \to \mathsf{U}_1 \to \mathsf{U}_2$; note than now the direction of arrows r_* is not reversed.

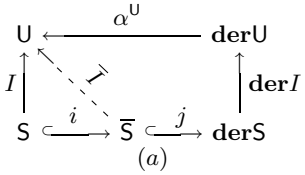
E.5 Queries and views. Normally, a model S specifies not the entire universe of discourse but a *generic* fragment of it. "Generic" here means that all the required information about the universe can be extracted from S-instances by applying to them certain predefined operations (queries, in the database terminology). So, in general, the repository $\mathcal{R}$ should be supplied with a collection of operations (query language), *Der*. This collection gives rise to a functor on the repository graph, $\mathbf{der}\colon \mathcal{R} \to \mathcal{R}$ (sect. A.4.1b), in the following way.

If a model is to be thought of as a syntactical schema (diagram, specification) S, then $\mathbf{der}\mathsf{S}$ is to be thought of as the augmentation of S with all possible derived items that can be specified by queries from *Der*. Then for any augmentation $\overline{\mathsf{S}}$ of S with a finite composition of queries, we have inclusions $\mathsf{S} \subset \overline{\mathsf{S}} \subset \mathbf{der}\mathsf{S}$. If a model is a semantic universe U, then $\mathbf{der}\mathsf{U}$ is to be thought of as the result of applying all possible queries to U, which should again belong to U. More accurately, there is a mapping $\alpha^{\mathsf{U}}\colon \mathbf{der}\mathsf{U} \to \mathsf{U}$ that assigns to each query (embodied into $\mathbf{der}$) its result. Existence of such an arrow means that the universe U is a $\mathbf{der}$-algebra (the construction is exactly dual to coalgebra described in D.1). Thus, we will assume that our repository graph

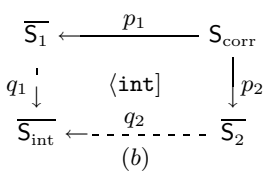
$\mathcal{R}$ is endowed with a functor **der** and all semantic universes (*ie*, nodes from class $\mathcal{U}$) are **der**-algebras.

Now, let $I\colon \mathsf{S} \to \mathsf{U}$ be an instance of S and $\overline{\mathsf{S}}$ is some augmentation of S with derived items, $\mathsf{S} \overset{i}{\hookrightarrow} \overline{\mathsf{S}} \overset{j}{\hookrightarrow} \mathbf{der}\mathsf{S}$. Then I can be extended to an instance $\overline{I}\colon \overline{\mathsf{S}} \to \mathsf{U}$ of $\overline{\mathsf{S}}$ by composition: $\overline{I} \overset{\text{def}}{=} j \fatsemi \mathbf{der}I \fatsemi \alpha^{\mathsf{U}}$ as shown on diagram (16a) where the trapezoid part is commutative by definition of $\overline{I}$ (and the derived arrow is dashed). In addition, we require also commutativity of the triangle: it means that the restriction of $\overline{I}$ to S equals to I since queries should not produce side effects. Thus, the commutative diagram (16a) models query mechanism in a very general and compact way.

(16)



(a)



(b)

Modeling view mechanism by arrows is now straightforward. A *view* to schema (model) S is a pair $V = (\mathsf{S}_V, v)$ with S_V the *view schema* and $v\colon \mathsf{S}_V \to \overline{\mathsf{S}}$ the *view morphism*, *ie*, a mapping of the view schema into some augmentation of S with derived items.[9] Any instance $I\colon \mathsf{S} \to \mathsf{U}$ of S gives rise to an instance I_V of S_V by composition of mappings: $I_V \overset{\text{def}}{=} v \fatsemi \overline{I}\colon \mathsf{S}_V \to \overline{\mathsf{S}} \to \mathsf{U}$.

E.6 Model integration. Integration of models S_1, S_2 according to some *correspondence information* is a special arrow diagram operation specified by the diagram (16b). On that diagram, the arrow span $(\mathsf{S}_{\text{corr}}, p_1, p_2)$ specifies the input, and the co-span $(\overline{\mathsf{S}_{\text{int}}}, q_1, q_2)$ is the output of the operation.

The model (schema) S_{corr} specifies overlapping of models S_1 and S_2 and the triple $(\mathsf{S}_{\text{corr}}, p_1, p_2)$ is, in fact, some correspondence relation between S_1 and S_2 (compare with sect. A.2. Note also that this correspondence may involve derived items of models – a point that seems to be missing in the literature on model integration). Model S_{int} is the result of integration modulo the correspondence above, and the mappings q_1, q_2 show how the component schemas are presented in the integrated one. The label $\langle$`int`] is a diagram operation marker denoting the procedure of integration; in fact, it is a particular case of familiar categorical operation of push-out. Details of this description can be found in [CD96].

E.7 Model repositories as sketches. A repository of models (of arbitrary nature) can be specified by a *model sketch* whose nodes denote models and arrows denote their mappings. Marked diagrams in this sketch denote statements about models, and manipulations with models are diagram operations over the model sketch (e.g., integration as considered above).

E.8 Heterogeneous MMt. What we have considered so far is related to managing models of the same type. However, an important issue in MMt is transformation of models, say, ER-diagrams into SQL code or UML-diagrams into Java. To capture this into our framework, we need to deal with mappings not *inside* a given specification space (repository) but *between* different specification spaces. Such mappings are quite familiar in CT under the name of *functors*, and they allow us to extend our arrow framework to capture heterogeneous MMt too (some details can be found in [Dis99]). A great advantage of the sketch language is that it allows reducing heterogeneous MMt to homogeneous MMt along the lines outlined in A.4.3(c).

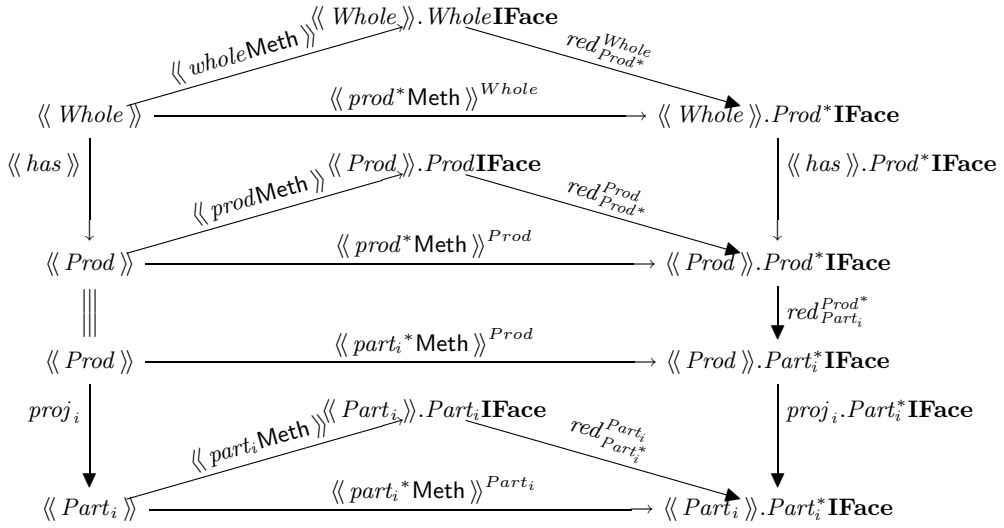


Figure 8: Behavioral aspects of composition.

Notes

[1]A Nobel laureate, physicist Eugene Wigner, once wrote a paper about "unreasonable effectiveness of mathematics" in natural sciences, specially devoted to that mystery. It may look indeed somewhat magical, but at least some light can be shed by noting that mathematics, natural sciences, and engineering, all exist in the same space of human culture, and are all made by humans sharing common intellectual and aesthetical (this also matters) criterions and patterns.

[2]As a rule, building an adequate formalization is an iterative procedure with essential feedbacks. That is, after some preliminary formal model of a concept is built, we may submit more reasonable and more precise questions to our intuition of the concept, rework the model and so on. On the other hand, new questions to intuition can change it: rearrange its aspects, reveal some important and even key point obscured or missed before, change the boundaries of the concept to be modeled and its relation to other concepts. Quite often, a good formal model can essentially change the intuitive perception of a complex concept. History of mathematics is full of such examples; and history of business modeling, according to [Kil99], demonstrates a similar phenomenon with "precise" substituted for "formal" as well.

[3]We need to be very careful when somebody claims that a formal semantics F for a concept/construct C is built. The first point to check is whether F is indeed formal or formalizable, since the cases when F is just a quasi-formal dress for pseudo-formal considerations are not rare in the literature. If F is indeed formal, we need to check its consistency; normally, in engineering applications, that is not a too difficult problem. What is much more non-trivial, and intriguing, is whether F is a really adequate formal model for C. Formally, this question is beyond formal models as such, but substantially is in the very heart of formal modeling.

[4]In CT, the term *sketch* denotes a format somewhat different from that one we use in this paper. Classical categorical sketches use only a special sort of diagram predicates denoting so called *universal properties*. Our sketches is a generalization towards much more flexibility of the sketch language. They were introduced by Makkai [**?**] and independently by the author [Dis97]. In addition to having arbitrary diagram predicates, sketches introduced in [Dis97]

also enjoy having arbitrary diagram operations.

[5]By their very nature, formal specifications have to concern about a lot of details of logical consistency, and in this sense go far into "formal depth". On the other hand, descriptions of engineering constructs have to concern about a lot of details of functioning in the real world, and in this sense cover a great "engineering width". So, formalization of engineering descriptions has to deal with a really spacious array of details, and leads to really unwieldy formal specifications unless effective means of compactifying them are used.

[6] This graphical similarity - between a complex UML diagram and a sketch specifying its formal meaning - might look somewhat magic, and the endnote 1 can be cited again. However, in the present case there are more direct technical causes for that. As we have seen, a system of object classes is semantically a fragment of some universe of behavior sets and mappings. This universe also carries certain diagram predicates and operations (though the latter are often underestimated in OO visual modeling, they are really important and often implicit in UML diagrams; as for databases, these operations are in the very heart – they are nothing but queries). On the other hand, CT (motivated by its own problems in algebraic geometry and logic) specially studied universes of objects that could be seen as far reaching generalizations of sets, varsets, behset and similar constructs. A rich theory of classifying and specifying such universes was developed (within the so-called *topos theory*). Following some terminological tradition, we will call an arbitrary universe of set-like formations closed under a certain set of operations a *Grothendieck's universe* (GU).

Thus, a particular universe of discourse usually dealt with in OO modeling can be considered as a finite piece of the corresponding GU. Then the general goal of UML developers can be formulated as follows: to build a system of diagrammatic notational tools for specifying (finite fragments of) wide range of GUs that matter in software engineering. Engineers have managed the task by inventing the UML diagrams. Categoricians were dealing with a formal refinement of basically the same task and invented sketches. Not surprising that the sketch format appears as a sort of formal refinement and abstract syntax for UML diagrams.

[7]Appendix E shows that specification aspects of MMt are heavily based on arrows (mappings between models, between semantic universes, between spaces of models and so on). On the other hand, in the paper [BHP00] written from positions much more close to practice, the notion of model mappings is also declared to be the major one in MMt. So, from either side, theoretical or practical, arrows in general, and model mappings in particular, are of fundamental importance for MMt. Interestingly, that the authors of [BHP00] don't even mention CT – a mathematical theory of arrow thinking, and try to reproduce some of categorical notions from scratch in a naive way. Nevertheless, all their considerations are reasonable and can be cleaned and formalized by CT-means.

[8]To model collective behavior, we just consider a collection of objects as a new system. Its states are tuples of component object states, and its methods are formed from (i) those composed from component object methods and (ii) new methods involving the component ones but not derivable from them.

[9]It is common to consider a view as just a subschema of $\overline{\mathsf{S}}$. Our arrow definition is much more flexible: it is reduced to the common definition when the view mapping is injective, but in general it is not necessary. In addition, the view mapping may evolve over a fixed view schema, see [**?**] for examples and discussion.

References

[BÖ2] E. Börger. The origins and the development of the ASM method for high level system design and analysis. *Journal of Universal Computer Science*, 8(1):2–74, 2002.

[BHP00] P. Bernstein, A. Halevy, and R. Pottinger. A vision for management of complex models. *SIGMOD Record*, 29(4):55–63, 2000.

[CD96] B. Cadish and Z. Diskin. Heterogenious view integration via sketches and equations. In *Foundations of Intelligent Systems, 9th Int.Symposium, ISMIS '96*, volume 1079 of *Springer Lect.Notes in AI*, pages 603–612, 1996.

[Dis97] Z. Diskin. Generalized sketches as an algebraic graph-based framework for semantic modeling and database design. Technical Report 9701, University of Latvia, 1997. `ftp://ftp.fis.lv/pub/diskin/ULReport M97.ps`.

[Dis99] Z. Diskin. Abstract metamodeling, I: How to reason about meta- and metamodeling in a formal way. In *8th OOPSLA Workshop on Behavioral Specifications, OOPSLA99*. Northeastern University, College of Computer Science, 1999.

[Dis00] Z. Diskin. On mathematical foundations for business modeling. In *TOOLS'37: 37th Int. Conference on Technology of Object-Oriented Languages and Systems. Sydney, Australia*, pages 182–187. IEEE Computer Society Press, 2000.

[Dis02a] Z. Diskin. Formal semantics for composition and aggregation, genearlizationa and specializationin relationships between object classess. Technical Report 0201, Frame Inform Systems/LDBD, Riga, Latvia, 2002.

[Dis02b] Z. Diskin. Visualization vs. specification in diagrammatic notations: A case study with the UML. In M. Hegarty, B. Meyer, and N. Narayanan, editors, *Diagrams'2002: 2nd Int. Conf. on the Theory and Applications of Diagrams*, volume 2317 of *Springer Lect.Notes in AI*, pages 112–115, 2002. Extended abstract. Full version is on `ftp://ftp.fis.lv/pub/diskin/Diagrams02.ps`.

[DK03] 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.

[GK01] Martin Gogolla and Cris Kobryn, editors. *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, volume 2185 of *Lecture Notes in Computer Science*. Springer, 2001.

[GLn01] G. Génova, J. Llorens, and P. Martínez. Semantics of the minimum multiplicity in ternary associations in UML. In *[GK01]*, pages 329–341, 2001.

[Gog93] J. Goguen. On notation. In *TOOLS 10: Technology of Object-Oriented Languages and Systems*. Prentice-Hall, 1993.

[Gog98] J. Goguen. An introduction to algebraic semiotics, with applications to user interface design. In C. Nehaniv, editor, *Computation for Metaphors, Analogy and Agents*, pages II: 54–79. Univertsity of Aizu, 1998. The latest version is available at `http://ww-cse.ucsd.edu/users/goguen`.

[JM02] Guest Editor J. Miller. Panel: What UML should be. *Communications of the ACM*, 45(11):67–85, 2002.

[JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebra and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[Kil99] H. Kilov. *Business specifications. The key to successful software engineering*. Prentice Hall PTR, 1999.

[Kob99] C. Kobryn. UML2001: A standardization Odyssey. *Communications of the ACM*, 42(10), 1999.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[Pol00] E. Poll. Coalgebraic semantics of subtyping. *Electronic Notes in TCS*, 33, 2000.

[Ste01] P. Stevens. On associations in the Unified Modeling Language. In *[GK01]*, volume 2185 of *Springer Lect.Notes in Comp.Sci.*, pages 361–375, 2001.

[Wol99] U. Wolter. A coalgebraic introduction to CSP. In B. Jacobs and J. Rutten, editors, *computer science. Electronic Notes in Theoretical Computer Science*, volume 19. Elsevier Science Publishers, 1999.

[WSW99] Y. Wand, V. Storey, and R. Weber. An ontological analysis of the relationship construct in conceptual modeling. *ACM TODS*, 24(4):494–528, 1999.