Formal Foundations of Metamodeling: Abstract Words, Abstract Languages, and Edge Algebra

Stefano Merenda, Markus Herrmannsdoerfer, and Martin Feilkas

Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
{merenda, herrmama, feilkas}@in.tum.de

Abstract. Textual languages are not the only way to write down models. Especially in the field of model-based engineering, graphical languages are gaining more and more popularity. In this regard well-known specification techniques for textual languages like context-free grammars are often substituted by metamodels. Nevertheless do we have to give up the theory of formal languages including their definitions for word and language if using metamodels for language specification.
In this paper we introduce a formal framework for the purpose of metamodeling by defining an abstract word as a special type of a partially ordered multi-graph (called M-graph). We consciously avoid a definition of a metamodeling language itself in order to show that this approach is independent of any metamodeling language. We will instead introduce an algebra on edges which allows to express complex consistency conditions on models as well as querying large models.

1 Introduction

Up to now a huge number of metamodeling approaches does exist, not to mention the related domains like databases, markup languages, and ontologies. All of them define their own metamodeling language. Nevertheless is the degree of formalization much lower than in formal languages. In order to discuss about semantics and expressiveness of metamodeling languages in a more sound way, we have to increase the degree of formalization. The crucial weak point of metamodeling is the lack of a *formal* and *appropriate* semantic domain. There is a set of formal approaches, like KM3 [1], but they are not appropriate for industrial practice. Important needs to mention are the support of orders and duplicates, uniqueness of canonical keys, context sensitive domains for properties (thus to restrict the set of possible values for a property), and an instantiation concpet. On the other hand appropriate approaches like MOF [2] are informal or the formalization is extremely complex, since these languages themselves provide a huge set of constructs. Nevertheless, in spite of the complexity MOF does not support all of the above listed needs.

Informally, a metamodel defines the set of models which are valid according to the given metamodel. This definition leads to the question about what is a

model. In most cases it is a graph-like structure, but the exact definition varies for each metamodeling approach. In contrast, the theory of formal languages has a very clear understanding of what a model is: It is called word and is simply defined as a sequence of terminal symbols, which are elements out of the so-called alphabet. On that basis it is easy to discuss about the definition of languages: A language is defined by a set of valid words.

Up to now there is no similar and suitable definition in the metamodeling domain. We therefore intend to propose the corresponding terms *abstract alphabet* and *abstract word* in order to allow for a clear definition of *abstract languages*. This is the reason why we are going to introduce a special kind of graph which will represent an abstract word. In other words we will define a semantic domain for metamodeling: Once we know the definition for an abstract word, we are able to discuss about how to define an abstract language (which is defined as a set of abstract words). To specify the semantics of a metamodeling language, we need to define for each abstract word whether it is specified by a dedicated metamodel within the abstract language or not.

As we will see, we strictly separate models (abstract words) from metamodels. Thus, a model can exist without any metamodel. Formal languages follows this paradigm in a natural way: We can write down a word based on a dedicated alphabet without knowing anything about the grammar. Afterwards we can discuss whether it is part of a given language or not. XML [3] follows the same paradigm: we can write down a XML document without having any schema definition. In contrast, most of the metamodeling approaches do not have this separation. The most explicit formulation for the dependency of models to metamodels is given in the formalization of KM3 in [1]: "A model $M = (G, \omega, \mu)$ is a triple where [...] ω is a model itself (called the reference model of M)." Thereby ω represents exactly the metamodel. Nevertheless there are some relevant reasons why a separation of models from metamodels is crucial:

1. *Support of bottom-up language engineering.* When specifying a new language, writing down a concrete model may often be the first step in order to get a clearer idea about the language to be created. Afterwards we specify the language definition (metamodel) itself. This procedure is only possible if the metamodel is not required to specify a model. This is also an advantage of XML and textual languages in general: Neither a XML schema nor a grammar are necessary for firstly defining models.
2. *Ability to correlate different metamodeling approaches.* Often, model data has to be transformed from one technical space as described in [4] to another. E.g. if we want to convert models stored within a object database into a XML-document for exchange reasons or we want to represent the same model by a textual language. In such cases we have to correlate the corresponding approaches for defining the structure of models like ODL [5], XML-Schema [6], and EBNF [7]. It is difficult to correlate metamodeling approaches without a generic formal framework for models. As a matter of course such a definition can not comprise metamodeling aspects, because then it could not be independent from a specific metamodel.

3. *Avoidance of recursive model definition.* When discussing metamodeling approaches different modeling levels are often introduced. In general, such definitions make sense, of course. Nevertheless such approaches lead to self-describing meta-metamodels and hence they cause a problem when we want to formally define the meta-metamodel by itself, i.e. by something we do not know yet since we are currently defining it.
4. *Appropriateness for metamodel evolution.* An important aspect in metamodel evolution is the migration of existing models. During the migration of models we have to deal with a switch from one metamodel to another. To talk about the state of the model during the migration between these two metamodels it is helpful to use a description of the model which is independent of both metamodels.

Up to now we have discussed the needs related to the *formality* of the semantic domain for metamodeling. Now we want to motivate the needs related to the *appropriateness* in industrial practice. We hereby want to address the gap between theory and practice. When analyzing the differences which are relevant for defining a semantic domain, the following two major issues are currently not sufficiently addressed by formal metamodeling approaches although they are necessary in real applications:

1. *Dealing with attributes and enumerations.* No practically feasible model can do without attributes. Most of them also rely on enumerations. We want to show how they can be mapped to our approach without explicitly extending it by such concepts. This guarantees an easy basic theory which covers also these additional but important concepts. All introduced operators for abstract words can also be applied to attributes and enumerations.
2. *Dealing with orders and duplicates.* A second issue of practically feasible metamodeling environments is the support for orders and duplicates. In MOF for example we can add $\{ordered\}$ and $\{bag\}$ to an association end in order to indicate that such an association is (totally) ordered and allows duplicates, respectively [8]. Formal approaches are mainly based on the set theory and thus do not take orders and duplicates into account. In contrast, our formal framework is based on partially ordered multisets (pomsets). As we will see, pomsets allow us to handle orders and duplicates as well as all combinations of them.

Altogether, we do not find an integrated and homogeneous formal framework, which can be used as suitable basis for formal metamodeling with relevancy to practice.

Outline. We will at first provide an overview of existing metamodeling approaches in Section 2. Before we are going to introduce abstract words in Section 4 we will give an overview of pomsets in Section 3. On that basis we will show in Section 5 how we can operate on abstract words and define predicates on abstract words by introducing the edge algebra. In Section 6 we define abstract languages in general and show how to use the edge algebra to define abstract languages. In Section 7 we conclude and provide directions for future work.

2 Related Work

Metamodeling is closely related to other technical spaces which also describe the structure of data. In the following section we point out the most important representatives which influence this work above-average. Summarized, they are the Relational Algebra [9], graph grammars [10], description logics [11], MOF [2] with its formalization KM3 [1], GME [12], and OCL [13].

The Relational Algebra has been primarily introduced by Edgar F. Codd in 1970 [9]. Till this day it forms the formal basis for most commercial database systems including their query language SQL [14]. At first, Codd defines a database state as a set of relations. Based on this, the algebra on relations allows to alter the database state with a small set of operators as well as to formulate consistency conditions. By doing so, the Relational Algebra majorly inspires this paper besides formal languages. In contrast to the Relational Algebra in our approach a model is described by a set of edge-functions instead of a set of relations. This re-definition allows us to easily solve the major weak points of the Relational Algebra that occur when adapting it for the metamodeling domain without loosing its simplicity: 1. Multi-valued attributes are fully supported, 2. a closure operator is provided, and 3. ordered sets and multisets are supported. Needless to say for each of the mentioned problems evolve extensions of the Relational Algebra over time. Object-oriented databases as formalized by Georg Gottlob, Gerti Kappel and Michael Schrefl in [15] allow multivalued attributes, Dar and Agrawal introduce a closure operator for SQL in [16], and Stephane Grumbach and Tova Milo extend the Relational Algebra to an algebra for pomsets in [17]. Nevertheless the resulting theories loose the impressive simlicity of the Relational Algebra. An exception is the mentioned work of Grumbach and Milo but in contrast to our approach they partially order the tuples while our approach bases on partially ordered edges which allows a one-to-one mapping to the metamodeling domain.

Graph grammars have been invented in the early seventies in order to generalize textual grammars [10]. As a consequence, they also advocate a strict separation between a graph and its corresponding grammar. However, graph grammars are more constructive by providing rules to produce all graphs belonging to a language. In contrast, our framework is more descriptive in the sense that it constrains the graphs belonging to a language. While allowing the specification of duplicates, graph grammars do not cater for orders.

Description Logics (DL) are a family of languages for knowledge representation to describe ontologies in a formally well-understood way. D. Nardi, R. J. Brachman, F. Baader, and W. Nutt give a detailed insight into DL in [11]. Most of the DLs are a decidable subset of first order logic which makes them attractive for inferring new knowledge from existing one. In contrast to our approach the expressiveness is much more restricted which makes it insufficient for metamodeling. DLs distinguish between a terminological box (tbox) and an

assertional box (abox). The first one describes the concepts of a domain, the second one knowledge about concrete instances. This distinction corresponds to that of metamodels and models in our domain. Beware that a concept in our approach is not the same as in DLs: In our approach each individual (node) is mapped to exactly one concept, while an individual may belong to many concepts in DL. Thus, a concept in DL can be seen as a node evaluation in our approach which makes it easy to use DLs in our approach.

The Meta Object Facility (MOF) [2] provides a standard for defining the abstract syntax of modeling languages. There are a number of realizations of MOF, like e.g. the Eclipse Modeling Framework (EMF) [18], which is probably the most widely used at present. As the MOF standard as such lacks a formal foundation, there have been a number of attempts to define a formalization. In [1], Jouault and Bezivin present a formal semantics of their textual language KM3 which addresses a subset of MOF called Essential MOF (EMOF). This formal semantics is based on Prolog and defines a number of predicates for nodes, properties and edges to relate a model to its metamodel. Poernomo presents a formalization of MOF [19] which is based on constructive type theory (CTT). This formalization is particularly suited to prove the correctness of metamodels through well-typedness. In [20], Boronat and Meseguer present an algebraic semantics for the MOF standard in membership equational logic (MEL). As they have operationalized this semantics within the Maude language, it can be used to perform formal analyses on models and metamodels. In contrast to these formalizations of MOF, we propose a formal framework for metamodeling in general which can be instantiated to formalize MOF.

The Generic Modeling Environment (GME) provides a metamodeling formalism for defining a modeling environment [12]. The underlying multi-graph architecture emerged from a generalization of component-based embedded systems [21]. Even though the origin of this architecture is quite formal, no up-to-date formalization of the approach exists. In addition, there is neither support for duplicates nor for orders.

The Object Constraint Language (OCL) provides a standard of an expression language to navigate and constrain models [13]. As a precise semantics is not part of the standard, there have been a number of attempts to formalize OCL. In [22], Brucker and Wolff propose a semantics for OCL based on a shallow embedding in Isabelle/HOL. In [23], Kyas et al. present a mapping of OCL constraints onto the PVS theorem prover. In [24], Markovi and Baar propose a formal semantics for OCL constraints based on their evaluation as QVT model transformations. While these formalizations map the OCL constraints to a separate semantic domain, our approach provides an algebra directly working on edges. In contrast to OCL, where constraints cannot be evaluated in a metamodel-independent way, the edge algebra does not require a metamodel to be present. In contrast to OCL the Edge Algebra is a very tiny language which makes it much easier to understand as well as to implement. Nevertheless due to its pomset support it is even more powerful than OCL in constraining bags and lists.

3 Pomsets in the Context of Metamodeling

Pomsets themselves have been known for many years and are used for process modeling in particular. Pratt introduced them in [25]. In [17] an algebra of pomsets is defined in order to create a generalization of the traditional algebras for (nested) sets, bags and lists. Nevertheless, in the domain of metamodeling they are not used until now, although sets, bags as well as lists have to be combined within one (meta-)model. To support this combination in a sound way, we will show that pomsets seem to be the right concept in the metamodeling domain, too. We will provide a short overview of pomsets in the following and introduce the operators which are important in the metamodeling domain.

3.1 Relationship between the different Types of Sets

As mentioned in Section 1, MOF introduces two modifiers required for sets: $\{ordered\}$ and $\{bag\}$. According to that, we are able to describe the four types *set*, *totally ordered set (toset)*, *bag*, and *list*. To formulate an appropriate mathematical foundation a common set type which generalizes all these four types is necessary. Unfortunately none of the four is a most general one. In particular, *Bag* generalizes *Set* (we write $Set \subset Bag$) and *List* generalizes *Toset* (we write $Toset \subset List$), but beside these two there is no further generalization relationship. To get the picture complete in a mathematical sound way, the *partially ordered multiset (pomset)* and its specialization *partially ordered set (poset)* has to be introduced. As we can see in Figure 1, *pomset* is a common generalization for all mentioned set types.

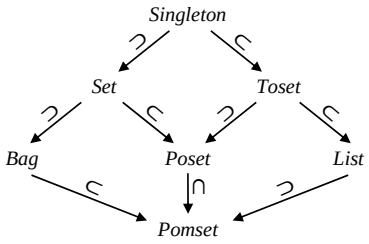


Fig. 1. Inclusion relationships between the different types of sets

We also depict *Singleton* in order to show that single valued *sets* and single valued *lists* are equal in the sense of pomsets. This correlates with the fact that for single-valued properties in MOF the modifiers $\{ordered\}$ and $\{bag\}$ are not allowed. Thus, in our approach there is no distinction between a single-valued set and a single-valued list. In addition to that we totally skip single values themselves. We always represent them as a single-valued pomset. For this reason, whenever a pomset is awaited, also a single value can be given, so we can write $a = \{a\}$.

3.2 Definition of Pomsets

As we could see in the previous sections, pomsets can provide us with a unified view of sets, lists as well as single values. Although pomsets are already known, we want to introduce them here again concentrating on a different focus: the metamodeling domain. Informally speaking, a pomset is a multiset where the elements are partially ordered in addition. If the partial order is *total* we get a so-called *totally ordered multiset (tomset)* which is also known as *list*. A formal definition for pomset is given as follows:

Definition 1 (Pomset). *A* partially ordered multiset (pomset) *is a quadruple* $\langle E, V, \lambda, \prec \rangle$ *where* E *is a basic set of elements,* V *a set of vertices,* $\lambda : V \to E$ *a total function which labels each vertex with an element out of the basic set, and* $\prec \subseteq V \times V$ *is a strict (irreflexive) partial order over* V*.*

Two pomsets $A = \langle E_A, V_A, \lambda_A, \prec_A \rangle$ *and* $B = \langle E_B, V_B, \lambda_B, \prec_B \rangle$ *are equal (denoted by* $A = B$*) if, and only if, there exists a bijection* $\varphi : V_A \rightarrow_{bij} V_B$ *such that* $\forall v \in V_A : \lambda_A(v) = \lambda_B(\varphi(v))$ *and* $\forall u, v \in V_A : u \prec_A v \Leftrightarrow \varphi(u) \prec_B \varphi(v)$*. The set of all partially ordered multisets over a basic set* E *is denoted by* $\mathcal{P}_{pomset}(E)$*.*

Note, that due to the given equivalence relation the concrete set of vertices is irrelevant. Thus, we are able to assume disjoint sets of vertices for any two pomsets if needed. Now are able to define the different types of sets based on the pomset definition. It is easy to verify that the relationships given in Figure 1 results from these definitions.

- $\mathcal{P}_{bag}(E) = \{\langle E, V, \lambda, \prec \rangle \in \mathcal{P}_{pomset}(E) \mid \prec = \emptyset\}$
- $\mathcal{P}_{list}(E) = \{\langle E, V, \lambda, \prec \rangle \in \mathcal{P}_{pomset}(E) \mid \prec \textit{ is total}\}$
- $\mathcal{P}_{poset}(E) = \{\langle E, V, \lambda, \prec \rangle \in \mathcal{P}_{pomset}(E) \mid \lambda \textit{ is injective}\}$
- $\mathcal{P}_{toset}(E) = \mathcal{P}_{list}(E) \cap \mathcal{P}_{poset}(E)$
- $\mathcal{P}_{set}(E) = \mathcal{P}_{bag}(E) \cap \mathcal{P}_{poset}(E)$
- $\mathcal{P}_{singleton}(E) = \{\langle E, V, \lambda, \prec \rangle \in \mathcal{P}_{pomset}(E) \mid |V| \leq 1\}$

3.3 Operators on Pomsets

We are going to define the pomset operators in the following section. Although some of these operators have already been defined in [17], we will quote those operators which turn out to be useful in the metamodeling domain and combine them with some new ones.

Definition 2 (Pomset operators). *Let* $A = \langle E_A, V_A, \lambda_A, \prec_A \rangle$ *and* $B = \langle E_B, V_B, \lambda_B, \prec_B \rangle$ *be two pomsets. We assume without loss of generality that* V_A *and* V_B *are disjoint.*

- ***Additive Union,*** $\uplus$***:*** *$A \uplus B$ is a pomset containing all the elements in A and B, which preserve the order of those elements and do not add any additional order. Thus, all elements in A are unrelated to those of B.*

$$A \uplus B =_{def} \langle E_A \cup E_B, V_A \cup V_B, \lambda_A \cup \lambda_B, \prec_A \cup \prec_B \rangle \tag{3.1}$$

- ***Concatenation,*** $\oplus$***:*** *$A \oplus B$ is a pomset containing all the elements in A and B, which preserve the order of those elements and additionally make all the elements in A smaller than those of B.*

$$A \oplus B =_{def} \langle E_A \cup E_B, V_A \cup V_B, \lambda_A \cup \lambda_B, \prec_A \cup \prec_B \cup (V_A \times V_B) \rangle \tag{3.2}$$

- ***Expansion,*** χ ***:*** *Let f be a function $f : E_A \to \mathcal{P}_{pomset}(E')$. $A \chi f$ is a pomset where all the elements $e \in A$ are replaced by the pomset $f(e)$, preserving the order. $f(e) = \langle E_{f(e)}, V_{f(e)}, \lambda_{f(e)}, \prec_{f(e)} \rangle$ denotes the pomset quadruple that results from applying f on $e \in E_A$.*

$$\begin{aligned} & A \chi f =_{def} \langle E', V', \lambda', \prec' \rangle \\ & \textit{where} \\ & V' = \{ (v, w) \mid v \in V_A \wedge w \in V_{f(\lambda_A(v))} \} \\ & \lambda' : V' \to E', (v, w) \mapsto \lambda_{f(\lambda_A(v))}(w) \\ & \prec' = \{ ((v, w), (v', w')) \mid (v \prec_A v') \vee (v = v' \wedge w \prec_{f(\lambda_A(v))} w') \} \end{aligned} \tag{3.3}$$

- ***Projection,*** $\downarrow$***:*** *$A \downarrow B$ is the pomset A in which all elements are removed that are not in B.*

$$\begin{aligned} & A \downarrow B =_{def} A \chi r \\ & \textit{where} \\ & r : E_A \to \mathcal{P}_{pomset}(E_A \cap E_B), e \mapsto \begin{cases} \{e\} & if e \in B \\ \emptyset & if e \notin B \end{cases} \end{aligned} \tag{3.4}$$

- ***Consists Of,*** $\Subset$***:*** *$A \Subset B$ if and only if all elements that occur in A also occur in B independant of the order and multiplicity of the elements.*

$$A \Subset B \Leftrightarrow_{def} A \downarrow B = A \tag{3.5}$$

- ***Cardinality,*** $|.|$***:*** *$|A|$ returns the total number of elements within the pomset A taking duplicates into account.*

$$|A| =_{def} |V_A| \tag{3.6}$$

- ***Multiplicity,*** $\in_m$***:*** *$a \in_m A$ returns the number of occurrences of a in the pomset A.*

$$a \in_m A =_{def} |A \downarrow \{a\}| \tag{3.7}$$

- ***Order Inverse,*** $\leftarrow$***:*** *$\overleftarrow{A}$ is the same pomset than A except for the order being inversed. If A is completely unordered, this operator has no effect.*

$$\overleftarrow{A} =_{def} \langle E_A, V_A, \lambda_A, \{(v, w) \mid w \prec_A v\} \rangle \tag{3.8}$$

- ***Order Destroy,*** μ***:*** *μA deletes the order of the pomset A. Thus, the result is a bag.*

$$\mu A =_{def} \langle E_A, V_A, \lambda_A, \emptyset \rangle \tag{3.9}$$

– ***Duplicate Destroy, ϵ:*** *ϵA is a pomset where all the vertices $v \in V_A$ that are mapped by λ_A to the same label collapse to one unique vertex with this label, and where the order on the new objects is the maximal order consistent with that of the sources of the elements. Thus, the result is a poset.*

(3.10)

$$\epsilon A =_{def} \langle E_A, E_A, id, \prec' \rangle$$

where

id is the identity

$$\prec' = \left\{ (e, e') \;\middle|\; \begin{array}{c} (\exists v, v' : (\lambda_A(v) = e) \wedge (\lambda_A(v') = e') \wedge (v \prec_A v')) \\ \wedge\ (\forall v, v' : (\lambda_A(v) = e \wedge \lambda_A(v') = e') \Rightarrow \neg(v' \prec_A v)) \end{array} \right\}$$

3.4 Used Notation

Having defined pomsets we are now going to introduce adequate notations for pomsets which we will use in the following sections.

Pomsets as quadruple. Using Definition 1, we can explicitly specify a pomset by the quadruple, like the following example:

$$\begin{aligned} A = \langle & \{a, b, c\}, \{v_1, v_2, v_3, v_4, v_5\}, \\ & \{(v_1 \mapsto a), (v_2 \mapsto a), (v_3 \mapsto b), (v_4 \mapsto c), (v_5 \mapsto b)\}, \\ & \{(v_1, v_2), (v_2, v_3), (v_2, v_4), (v_1, v_3), (v_1, v_4)\} \rangle \end{aligned} \tag{3.11}$$

Since the $\prec$-relation has to be transitive, tuples which result from transitivity can be omitted. Hence we can define orders by *transitively reduced relations*. It is defined as the smallest subset of the relation at which its transitive closure results in the original relation. In our example (3.11) we are now able to skip the last two tuples (v_1, v_3) and (v_1, v_4).

Pomsets as transitive reduced graphs. Here we now want to introduce a more intuitive notation for pomsets which is based on graph representations and is closer related to the basic idea of sets. A pomset is represented by a directed graph which represents the transitively reduced partial order. The nodes are labeled with the pomset elements. Now we can specify our example (3.11) by:

$$A = \left\{ \begin{array}{ccc} \mathsf{b} & & \\ \mathsf{a} \rightarrow \mathsf{a} & \searrow & \mathsf{b} \\ & & \mathsf{c} \end{array} \right\} \tag{3.12}$$

Bags specified by the multiplicity function. Finally, we want to extend a common notation defining sets in order to define bags: By $B = \{e \in E \mid m(e)\}$ a bag is defined at which the function $m : E \to \mathbb{N}$ is exactly the multiplicity function of the specified bag B.

4 Models as Abstract Words

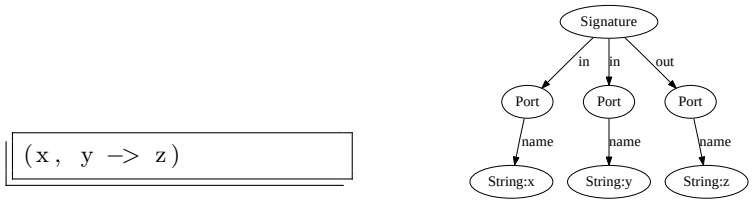
In order to formalize metamodeling languages e.g. for comparing two different types of them, we need to introduce models first. Models are often defined as *instances of metamodels*. Independent of the concrete meaning of what an instance is, this definition leads to the undesired strong coupling between model and metamodel. Instead of that, we follow the idea of formal languages: The definition of the principal superstructure of words (a sequence of terminal symbols) is completely independent of the way of defining the language. Influenced by that idea, we will define a general superstructure for models independent of a metamodeling language. We will call a model defined in such a superstructure an *abstract word*. Whereas words, as defined in formal languages, represent a model in a concrete textual way (for a better distinction we will call them *textual concrete words*), abstract words concentrate on the underlying *concepts* including their *properties* which describe the links in between. In the following section we will provide a formal definition of these abstract words.

4.1 Definition of Abstract Words

While textual concrete words are defined as a sequence of symbols out of an alphabet, an abstract word will be defined by a special type of a directed graph, where the nodes are labeled by *concepts* and the directed edges are labeled by *properties*. Thus, the basic elements for abstract words are not terminal symbols but a set of concepts and properties. According to that, we define an *abstract alphabet* as follows:

Definition 3 (Abstract alphabet). *An* abstract alphabet Σ *is a tuple* $\langle C, P \rangle$ *where* C *is a set of* concepts *and* P *is a set of* properties*. Furthermore, the two sets* C *and* P *must be disjoint.*

For a better readability we write concepts always starting with a capital letter, and properties with a small letter. Before we give the complete formal definition of an abstract word we want to introduce a short example which describes a (simplified) signature of a component as defined in the Component Language (COLA) [26]:



(a) textual concrete word (b) abstract word

Fig. 2. Example: abstract and textual concrete word of a signature

Especially when looking at the alphabets, the difference can be seen: For the textual concrete word the (minimal) alphabet of terminal symbols is $\Sigma = \{$ '(', ')', ',', '−', '>', 'x', 'y', 'z', ' '$\}$, the (minimal) abstract alphabet for the abstract word is $\Sigma = \langle \{Signature, Port, String\}, \{in, out, name\} \rangle$. While the textual concrete word is just a collection of characters, the abstract word shows much more details. Even a taxonomy can be derived.

In our example the order of the input ports is not reflected in the abstract word. For this reason outgoing edges of a node labeled with the same property should be ordered optionally. In general, the outgoing edges are described as a pomset of the target nodes:

Definition 4 (Edge function). *An edge function over a set of vertices V is a total function which maps each vertex to a pomset of target vertices. The set of all edge functions is denoted by $\mathcal{E}_V = V \to \mathcal{P}_{pomset}(V)$.*

Having edge functions we are able to formally define the superstructure for graphs like given in Figure 2(b):

Definition 5 (M-graph). *A* model graph (M-graph) *is a Σ-labeled, directed, partially ordered multigraph which is described as a quadruple $\langle \Sigma, V, type, edge \rangle$ where*

- *$\Sigma = \langle C, P \rangle$ is an abstract alphabet,*
- *V is a set of nodes which is disjoint to both sets C and P out of Σ,*
- *$type : V \to C$ is a total function which labels each node with a concept out of the alphabet, and*
- *$edge : P \to \mathcal{E}_V$ is a total function which assigns to each property an edge function out of $\mathcal{E}_V$.*

For a given M-graph a directed multigraph $\langle V, E \rangle$ with the identical set of vertices can be derived by defining the multiset E as follows:

$$E =_{def} \left\{ (v_1, v_2) \in V \times V \;\middle|\; \sum_{p \in P} v_2 \in_m edge(p)(v_1) \right\} \tag{4.1}$$

Note that due to the mapping to directed multi-graphs we are able to adopt the terms and definitions – like e.g. *cycle* and *path* – from graph-theory to M-graphs.

Sometimes we need the inverse *type*-function which is defined as:

$$type^{-1} : C \to V,\ c \mapsto \{v \in V \mid type(v) = c\} \tag{4.2}$$

According to Definition 5 we can write the graph in Figure 2(b) also as a quadruple $\omega = \langle \Sigma, V, type, edge \rangle$ where $\Sigma = \langle C, P \rangle$ and

- $C = \{Port, Signature, String\}$
- $P = \{in, out, name\}$
- $V = \{s, p_1, p_2, p_3, x, y, z\}$

$$- \ type = \left\{ \begin{matrix} (s \mapsto Signature), (p_1 \mapsto Port), (p_2 \mapsto Port), (p_3 \mapsto Port), \\ (x \mapsto String), (y \mapsto String), (z \mapsto String) \end{matrix} \right\}$$

$$- \ edge = \left\{ \begin{matrix} \left(in \mapsto \left\{ \begin{matrix} (s \mapsto \{p_1 \rightarrow p_2\}), (p_1 \mapsto \emptyset), (p_2 \mapsto \emptyset), (p_3 \mapsto \emptyset), \\ (x \mapsto \emptyset), (y \mapsto \emptyset), (z \mapsto \emptyset) \end{matrix} \right\} \right), \\ \left(out \mapsto \left\{ \begin{matrix} (s \mapsto \{p_3\}), (p_1 \mapsto \emptyset), (p_2 \mapsto \emptyset), (p_3 \mapsto \emptyset), \\ (x \mapsto \emptyset), (y \mapsto \emptyset), (z \mapsto \emptyset) \end{matrix} \right\} \right), \\ \left(name \mapsto \left\{ \begin{matrix} (s \mapsto \emptyset), (p_1 \mapsto \{x\}), (p_2 \mapsto \{y\}), (p_3 \mapsto \{z\}), \\ (x \mapsto \emptyset), (y \mapsto \emptyset), (z \mapsto \emptyset) \end{matrix} \right\} \right) \end{matrix} \right\}$$

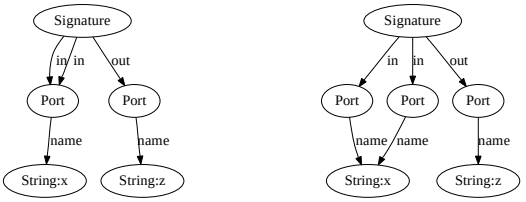
As usual in the context of graph theory, the concrete set of vertices is irrelevant. Thus, we define the equivalence of M-graphs as follows:

Definition 6 (M-graph equivalence). *Two M-graphs* $B = \langle \Sigma_1, V_1, type_1, edge_1 \rangle$ *and* $\omega_2 = \langle \Sigma_2, V_2, type_2, edge_2 \rangle$ *are equal (denoted by* $\omega_1 = \omega_2$*) if, and only if, there exists a bijection* $\varphi : V_1 \rightarrow_{bij} V_2$ *such that* $\forall v \in V_1 : type_1(v) = type_2(\varphi(v))$, $\forall p \in P_1 \cap P_2 : v \in V_1 : edge_1(p)(v) \, \chi \, \varphi = edge_2(p)(\varphi(v))$, $\forall p \in P_1 \setminus P_2 : v \in V_1 : edge_1(p)(v) = \emptyset$, *and* $\forall p \in P_2 \setminus P_1 : v \in V_2 : edge_2(p)(v) = \emptyset$.

Note, that φ can be used as a replacement function for the pomset expansion operator since we uniquely treat single values and singleton pomsets as introduced in Section 3.1.

According to this definition, unused concepts and properties can be added to the alphabet without having an impact on the equivalence relation. This characteristic of the abstract alphabet is similar to that of formal languages: A word does not change when adding additional (unused) symbols to the alphabet.

According to Definition 5, a property may contain duplicates, because the co-domain of edge functions is $\mathcal{P}_{pomset}(V)$. Since edges are defined over the set of vertices, duplicates in the resulting pomset of an edge-function occur if and only if there exist multiple edges from one node to another labeled by the same property. In Figure 3(a) a signature with a duplicate port is shown. In Figure 3(b) the same signature is modeled but without having multiple edges from signature to one port:



(a) Signature with duplicate port

(b) Signature with equal ports

Fig. 3. Example: Signature modeled with duplicate and same port

Anyway, the signature in Figure 3(b) has duplicate ports, but we are not able to dedect them by just inspecting a single pomset. Note, that this problem does not occur until pomsets or bags are introduced. When ignoring this problem, sets may also contain duplicates. To get rid of said problem in abstract words we only allow M-graphs which do not contain such clones in them. In order to achieve a formal definition we introduce *reachable sub-graphs*, *node equivalence*, and *minimal M-graphs*.

The idea behind this definition is that two nodes are equal if the sub-graph the nodes can "reach" is equal. The definition of "reaching" something is simply that part of the M-graph we can navigate to transitively. For example in Figure 3 the signature can reach everything while a port can not reach the signature or the other port. Since the sub-graph the two ports in Figure 3(b) can reach is exactly the same they should be seen as equal. Formally we define:

Definition 7 (Reachable sub-graph). *For a given M-graph* $B = \langle \Sigma, V, type, edge \rangle$ *and a node* $v \in V$ *we define the reachable sub-graph for* v *in* ω *as* $\mathbf{rsub}_\omega(v) =_{def} \langle \Sigma, V_R, type \cap (V_R \to C), edge \cap (P \to \mathcal{E}_{V_R}) \rangle$ *where* V_R *is the set of all reachable nodes starting from* v*, thus* $V_R = \left\{ v' \in V \;\middle|\; \text{it exists a path } v \xrightarrow{*} v' \right\}$.

Based on this, we are able to define the node equivalence:

Definition 8 (Node equivalence). *Given a M-graph* $\omega = \langle \Sigma, V, type, edge \rangle$*, two nodes* $v_1, v_2 \in V$ *are equal, if and only if there exists a bijection* φ *representing a M-graph equivalence between the two reachable subgraphs, such that* $\mathbf{rsub}_\omega(v_1) = \mathbf{rsub}_\omega(v_2)$*, and the bijection* φ *for this M-graph equivalence maps* v_1 *to* v_2*, thus* $\varphi(v_1) = v_2$.

Note, that according to these definitions, two equal nodes may also see each other, like in Figure 4.

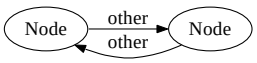


Fig. 4. Example: Equal nodes seeing each other

Since we want to exclude M-graphs having equivalent nodes, we define abstract words as minimal M-graphs:

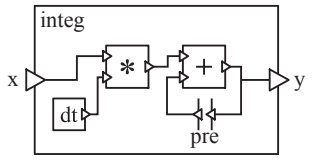
Definition 9 (Minimal M-graph). *A M-graph is called minimal, if and only if there do not exist two different nodes that are equal.*

Definition 10 (Abstract word). *An abstract word is a minimal M-graph.*

Note, that minimality can be easily substituted by a different equality operator for vertices. In our formal definition this would immediately lead to minimal M-graphs because a *set* of vertices is not able to contain *two* vertices that are equal.

4.2 Running Example

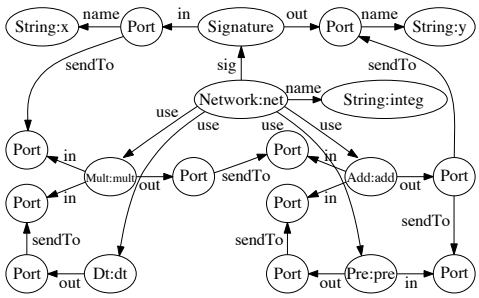
In order to get a more distinct picture, we will introduce a little running example out of the Component Language (*COLA*) [26]. COLA is an integrated modeling language for embedded reactive systems providing specification techniques for requirements, system design, implementation, and test. The language has been developed for the automotive industry and its specification is completely based on the theoretical foundation given in this paper. Through *OOMEGA* [27] this formal language specification is also used to generate an eclipse-based [28] tool architecture including a common database back-end. Since COLA is an extensive language we want to concentrate on a very small and simplified part of COLA in our example: the data flow networks. In this context it is not important to go into details of COLA. We will therefore give a short and informal explanation of what our example does. In order to get a clear understanding of the syntax and semantics of COLA we refer to [26]. Figure 5 shows an implementation of an integrator over time modeled in COLA. One and the same model is shown in three different ways: the first one shows the model as a graphical concrete word, the second one as a textual concrete word, and the third one as an abstract word.



(a) Graphical concrete word

```
network integ (x -> y)
{
  y := x*dt()+pre(y);
}
```

(b) Textual concrete word



(c) Abstract word

Fig. 5. Example: graphical, textual, and abstract word of a COLA model

4.3 Used Notation

As we can see in Figure 5(c), we use a traditional graph notation for visualizing abstract words. The edges are labeled by properties. If The label of each node always shows the assigned concept. If the vertex out of V is of interest to a node,

it can optionally be shown after the concept separated by a colon. E.g. the node labeled by $String : x$ implies that the concept is $String$ and the vertex element is x. Strictly speaking, the vertex element x has nothing to do with the fact, that this element should represent a string "x". In fact, the two nodes $String : x$ and $String : y$ from the example are equal according to Definition 8 since the concrete vertex element is irrelevant. Thus, without additional explanation this M-graph is even not an abstract word since it is not minimal. Essentially, this notation is a shortcut in order to avoid an explicit modeling of the strings "x", "y" or "integ". How this can be done in our framework is shown in Section 4.4.

4.4 Modeling dedicated Constructs by Abstract Words

In the following section, we want to outline how to model usual constructs of the metamodeling domain by abstract words.

Bidirectional associations Bidirectional associations in metamodeling represent links between nodes that are navigable in both directions. We can for example assume for the *sendTo* property of our example model an additional property *receiveFrom*, that represents the other direction of the association. Since edges in M-graphs are always directed, we represent a bidirectional link by two separate edges. Figure 6 illustrates a bidirectional link between to ports.

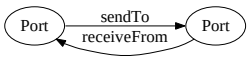
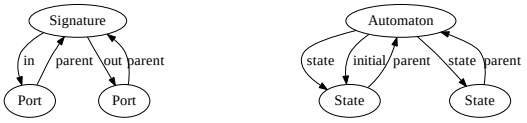


Fig. 6. Example: Bidirectional link

Note, that two properties like these are not marked as being the opposite of each other. Also the condition that for each property exists an opposite one is not guaranteed at the level of abstract words. Note, that modeling a link bidirectionally has influence to the node equivalence of Definition 8.

Compositions are a special kind of association where one node *is part of* another one. A port can for example be seen as a part of a signature. In order to express this fact in an abstract word explicitly, a special property *parent* is introduced. If a node is part of another one, an edge labeled with this *parent* property points to the containing node. Figure 7(a) illustrates a signature which contains two ports. By this approach the compositional link becomes an explicit part of abstract words. As for associations the specific characteristics of compositional links (e.g. that a node must have at least one parent) are not specified at the level of abstract words. Note, that this way of modeling compositional links does not allow to bind the composition to a specific property. Imagine an automaton which contains two states and one is marked as the initial state by an additional edge labeled with the property *initial* as shown in the Figure 7. It can not be decided whether the compositional property is *state* or *initial*.



(a) Unique (b) Not unique

Fig. 7. Example: Compositional links

Attributes In our examples we have already used some attributes like character strings. Another practically relevant example are numbers. Up to now we only introduced the shortcut notation for attributes. Most metamodeling approaches introduce a set of primitive types in order to solve this requirement. In order to prevent the need of additional theories, we want to demonstrate in this section that attributes themselves can be described by the already introduced constructs without any primitive type. Thus, this approach also allows a flexible treatment of the set of primitive types.

In principle, types whose domain is a finite set, can be modeled by adding a concept for each value of the type. In general, enumerations as used in metamodeling are of that kind. A relevant example is the type *Boolean*: For each logical value an own concept is added to the abstract alphabet: $True$ and $False$. Thus we can model the initial state out of our example in Figure 7 in a different way in Figure 8.

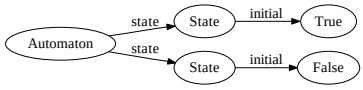


Fig. 8. Example: Boolean values in abstract words

For types whose domain is an infinite set like for natural numbers or strings this procedure is not sufficient. Since most data structures can be modeled by natural numbers we show how to model them: Each node representing a number greater than zero simply points to its predecessor whereas zero has no predecessor. Figure 9 models the numbers up to three.

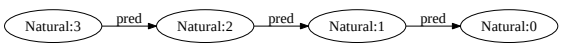


Fig. 9. Number defined by predecessor

Based on natural numbers all the data structures can be created. Strings for example can be modeled as a sequence of characters. A character itself is modeled as a natural number which represents the unicode.

5 Constraints on Abstract Words - the Edge Algebra

In the previous section it has been shown how it is possible to describe models as abstract words in the form of M-graphs, which has solely been introduced for that purpose. In model-based engineering it is now necessary in many situations to be able to infer new properties from the basis of a given model or to check consistency conditions. Analogous to relational data bases, a query and constraint language is essential in both cases. Whereas the well known Relational Algebra [9] offers a formal basis for SQL [14], a corresponding suitable approach for the metamodeling domain is still missing. The Edge Algebra presented herein is supposed to bridge this gap.

In our example out of Figure 5(c) one might put the question which unit depends directly or indirectly on others. The answer would come in the form of a newly calculated edge function *dependsOn*. The result is intuitionally easy to indicate: E.g. *add* depends directly on *pre* and *mult*, which means that $dependsOn(add) = \{pre, mult\}$. We might also ask what the correct execution orders of the units in the network are for, which leads to the causal order of the units. We will show that since we are able to deal with partial orders we can also express such computations considering orders and duplicates in an elegant way.

We will introduce the Edge Algebra in two steps: At first, we will concentrate on the basic Edge Algebra which exclusively focuses on navigation across edges. Secondly, we will extend the Edge Algebra in order to be able to deal with predicates, which allows to describe consistency conditions.

5.1 Basic Edge Algebra

Carrier set As already the name reveals, the edge algebra forms an algebra across edges as defined in Section 4 for abstract words. More precisely, the algebra is defined across edge functions $V \to \mathcal{P}_{pomset}(V)$ to a given set of vertices V. The carrier set is therefore defined as follows:

Definition 11 (Carrier set for the basic Edge Algebra). *The carrier set of the basic edge algebra is the set of all edge functions $\mathcal{E}_V$.*

From our example in Figure 5(c) all edges labeled by one property like *sendTo* or *use* would correspond to exactly one element of this carrier set. The Edge Algebra now defines operators to deduce new edge functions (such as *dependsOn*) from given edge functions (like *sendTo* and *use*).

From abstract words to edge-functions: The Edging Operator The alert reader will have noticed that the properties *sendTo* and *use* do, however, correspond to one edge function respectively, but according to the definition of abstract words they are at first only an identifier for elements from the set of properties P. Moreover, individual vertices from V and concepts from C respectively cannot be used directly in the edge algebra as they do not dispose of the form of an edge function $V \to \mathcal{P}_{pomset}(V)$. In the following, concepts,

properties as well as vertices are supposed to be transferred in edge functions for the edge algebra. This transition can be compared to that of an E/R model to a mere relational model. While therein the two constructs *entity* and *relation* are reduced to one construct *relation*, herein the constructs *concept*, *property* (which directly correspond to an edge function) and *vertex* are also reduced to a single construct *edge function*. Therefore we introduce the edging operator:

Definition 12 (Edging Operator). *For a given M-graph* $\omega = \langle \Sigma, V, type, edge \rangle$ *where* $\Sigma = \langle C, P \rangle$ *and the set of vertices* V *is disjoint to both* C *and* P*, we define the edging operator as follows:*

$$x^{\omega} =_{def} \begin{cases} edge(x) & \text{if } x \in P \\ V \to \mathcal{P}_{pomset}(V), v \mapsto type^{-1}(x) & \text{if } x \in C \\ V \to \mathcal{P}_{pomset}(V), v \mapsto \{x\} & \text{if } x \in V \\ V \to \mathcal{P}_{pomset}(V), v \mapsto \emptyset & \text{otherwise} \end{cases} \tag{5.1}$$

The edging operator thus maps the identifiers for concepts, properties and vertices in the context of a given abstract word to edge functions. It is hereby differentiated between four cases:

1. As already indicated, *properties* are mapped one-to-one to edge functions. If our exemplary model is given by ω, consequently $sendTo^{\omega}$ and use^{ω} would be the corresponding edge functions.
2. *Concepts* are mapped to edge functions that return for each vertex the set of all vertices that are labeled by the given concept. Since this resulting set is independent from the given vertex these edge functions are called constant. An example from our abstract word is $Port^{\omega}$. The resulting edge function points from every vertex to those vertices that are labeled with the concept $Port$.
3. *Vertices* are mapped to constant edge functions as well. The result will be for each vertex a single-valued set with the respective vertex. An example is $integ^{\omega}$ which points from every vertex to the vertex *integ*.
4. In all other cases (the identifier is not contained in any of the three sets C, P or V) the resulting edge function assigns the empty set to every vertex. Thus, no vertices are connected by this edge function. It is thus possible to apply any identifier at the edging operator without demanding them explicitly in one of these sets.

If it can be seen from the context that an edge function is expected and which abstract word is refered to, the identifier out of C, P or V can be used as an edge function without denoting the superscript identifier for the abstract word.

Additionally we are going to introduce the Star Edging Operator which is defined as the additive union of all properties of the abstract alphabet. Here as well the superscript ω may be omitted in case it already becomes obvious from the context:

Definition 13 (Star Edging Operator). *For a given M-graph* $\omega = \langle \Sigma, V, type, edge \rangle$ *we define the star edging operator as follows:*

$$*^{\omega} =_{def} V \to \mathcal{P}_{pomset}(V), v \mapsto \biguplus_{p \in P} edge(p)(v) \tag{5.2}$$

This definition becomes decisive if assertions are to be described about the totality of all properties defined in the abstract word since we do not have universal quantifiers $\forall$. An example for such a condition is avoiding *all* properties except *sendTo* for each *Port*.

Derived pomset operators Having now described how a given abstract word is described as a set of edge functions, we will now address the operators of the algebra. Most of the operators can be canonically deduced from the pomset operators. This is achieved by applying the pomset operator for each vertex to the resulting pomsets of the given edge functions:

Definition 14 (Derived pomset operators).

$$\begin{aligned} &op : {\mathcal{E}_V}^n \quad \to \mathcal{E}_V \\ &\quad e_1, \ldots, e_n \mapsto op(e_1, \ldots, e_n) =_{def} \begin{pmatrix} V \to \mathcal{P}_{pomset}(V), \\ v \mapsto op(e_1(v), \ldots, e_n(v)) \end{pmatrix} \\ &\textit{where} \\ &\quad op \in \left\{ \uplus, \oplus, \downarrow, \overleftarrow{\cdot}, \mu, \epsilon \right\} \\ &\quad n \textit{ is the arity of the operator } op \end{aligned} \tag{5.3}$$

It must be taken into consideration that the elements of the carrier set $\mathcal{E}_V = V \to \mathcal{P}_{pomset}(V)$ are functions themselves. This primarily results in the somehow unfamiliar definition of the operators which is to be explained in short by means of the additive union: For two edge functions $e, f \in \mathcal{E}_V$ the resulting edge function given by $e \uplus f$ maps each vertex $v \in V$ to $e(v) \uplus f(v)$.

The four core operators: Reflexive, Inverse, Closure, and Navigation Besides the canonically defined operators there are four additional operators which are defined in the following section.

Definition 15 (The core edge operators). *Let* $e, f \in \mathcal{E}_V$ *be two edge functions. The four operators are defined as follows:*

- ***Reflexive:*** $self$ *is a constant edge operator whose resulting edge function points for each vertex to itself.*

$$\begin{aligned} self : &\to \mathcal{E}_V \\ &\mapsto self =_{def} (V \to \mathcal{P}_{pomset}(V), v \mapsto \{v\}) \end{aligned} \tag{5.4}$$

- ***Inverse:*** *The inverse of an edge function inverts the directions of all edges. Hereby the order of the pomsets is lost since only the outgoing (and not the incoming) edges of one vertex can be ordered. Duplicates in turn remain. A double inversion therefore results in the initial edge function, however without order.*

$$\begin{aligned} {}^{-1} &: \mathcal{E}_V \to \mathcal{E}_V \\ e \quad &\mapsto e^{-1} =_{def} (V \to \mathcal{P}_{pomset}(V),\, v \mapsto \{w \in V \mid v \in_m e(w)\}) \end{aligned} \tag{5.5}$$

- ***Navigation:*** *The navigation $e.f$ forms that edge function that results if one firstly navigates along e and then along f, i. e. if the edges described by e are composed with those of f. By doing so, the resulting edge function will contain duplicates if there exist multiple paths from one vertex to another via e followed by f. Also the order is preserved. If one of the operators is ordered and the other not, the result will be partially ordered. Formally the navigation is defined by the expansion operator:*

$$\begin{aligned} . &: \mathcal{E}_V \times \mathcal{E}_V \to \mathcal{E}_V \\ e, f \quad &\mapsto e.f =_{def} (V \to \mathcal{P}_{pomset}(V),\, v \mapsto e(v) \,\chi\, f) \end{aligned} \tag{5.6}$$

- ***Closure:*** *As usual definitions, the closure ${}^{\wedge}e$ for a given edge function e maps each vertex to all vertices that are transitively reachable by a vertex including itself. Additionally the resulting set of vertices may be partially ordered: At first, the order of e is preserved. At second, the vertices will be brought in an ascending order via the length of the paths to reach these vertices. If the order of two vertices is contradictory or if there exists a directed cycle, the two vertices are unordered. Formally the closure is inductively defined:*

$$\begin{aligned} &{}^{\wedge} : \mathcal{E}_V \to \mathcal{E}_V \\ &\quad e \quad \mapsto {}^{\wedge}e =_{def} \lim_{n\to\infty} {}^{\wedge}_{n} e \\ &\textit{where} \\ &\quad {}^{\wedge}_{0} e =_{def} self \\ &\quad {}^{\wedge}_{n} e =_{def} \epsilon \left({}^{\wedge}_{n-1} e.\,(self \oplus e) \right) \end{aligned} \tag{5.7}$$

Running Example As described at the beginning of this section we can describe new edge functions based on given ones. At this point we want to demonstrate how the basic edge algebra can be used:

- $receiveFrom = sendTo^{-1}$ describes where the data comes from.
- $simpleDependsOn = in.receiveFrom.out^{-1}$ describes on which operators an operator depends on directly.
- $executionOrder = sig.out.receiveFrom.out^{-1}.\overleftarrow{{}^{\wedge} simpleDependsOn}$ describes the execution order of the operators for a network. In this definition the specialty of the $pre()$ operator is not taken into account which will result in

the fact that *Add* and *Pre* are not ordered. Later on we will present a correct definition using the extended Edge Algebra introduced in the following section. For the vertex *net* it will evaluate to

$$executionOrder(net) = \left\{ \begin{array}{l} \mathsf{dt} \rightarrowtail \mathsf{mult} \rightarrowtail \mathsf{add} \\ \mathsf{pre} \end{array} \right\}$$

5.2 Extended Edge Algebra

As shown in the above mentioned examples, we are able to deduce new edge functions from a given set of edge functions and are thus able to calculate new properties. Another important aspect represents the definition of consistency conditions on abstract words. Such consistency conditions in the scope of our exemplary model from Figure 5(c) were for example:

- A network must have a signature.
- Ports of a signature must have a name.
- The vertices with the concept $Mult$ and *Add* must have at least two input ports and exactly one output port.

The edge algebra will in the following be extended in order to be able to formulate and evaluate such predicates.

Extended carrier set In a first step the carrier set of the algebra will therefore be extended by so-called node predicates and node valuations. The extended edge algebra therefore now comprises the following three carrier sets in total:

$$\begin{aligned} \mathcal{E}_V &=_{def} V \rightarrow \mathcal{P}_{pomset}(V) \\ \mathcal{B}_V &=_{def} V \rightarrow \mathbb{B} \\ \mathcal{N}_V &=_{def} V \rightarrow \mathbb{N} \end{aligned} \tag{5.8}$$

Accordingly, node predicates assign a logical value $true$ or $false$ to each vertex, whereas the node valuation assigns a natural number (including zero) to each vertex.

Derived boolean and numeric operators Having extended the carrier set, we will introduce the corresponding additional operators in the following. Similar to the pomset operators, all operators – except for the selection and generality – can be canonically transfered here as well. The basic principle is – just as in the case of the pomset operators – that operators are applied to the evaluations of each vertex.

Definition 16 (Derived extended edge operators).

(5.9)
$$\begin{aligned} op : {\mathcal{X}_V}^n &\to \mathcal{Y}_V \\ x_1, \ldots, x_n &\mapsto op(x_1, \ldots, x_n) =_{def} \begin{pmatrix} \mathcal{Y}_V, \\ v \mapsto op(x_1(v), \ldots, x_n(v)) \end{pmatrix} \end{aligned}$$

where

$op \in \{\Subset, =, |\cdot|\} \cup \{\leq, =, +\} \cup \{\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\}$

n *is the arity of the operator* op

$\mathcal{X}_V, \mathcal{Y}_V \in \{\mathcal{E}_V, \mathcal{B}_V, \mathcal{N}_V\}$ *corresponds to the domain and co-domain of* op

The set of operators is divided into three subsets which should indicate the three types of operators: pomset, numerical, and boolean operators. Note, that the equality operator (=) can be applied to both pomsets and numericals.

Extended core operators: Selection, and Generality In the following we are going to define two more operators. The first one strengthens the correlation between node predicates/valuations and edge functions. The second one defines an universal quantifier over all nodes in order to specify global conditions.

Definition 17 (The extended core operators). *Let* $p \in \mathcal{B}_V$ *be a node predicate and* $n \in \mathcal{N}_V$ *a node valuation over the set of vertices* V*. The two operators are defined as follows:*

- ***Selection:*** σp *results to a constant edge function that maps each vertex to the (same) set of all vertices for which the given node predicate evaluates to true. In case of a node valuation instead of a set a bag is returned.*

(5.10)
$$\begin{aligned} \sigma : \mathcal{B}_V \cup \mathcal{N}_V &\to \mathcal{E}_V \\ x &\mapsto \sigma x =_{def} \left(V \to \mathcal{P}_{pomset}(V),\ v \mapsto \{w \in V \mid x(w)\}\right) \end{aligned}$$

- ***Generality:*** $\mathcal{G}^\omega p$ *evaluates to true if and only if the given node predicate evaluates to true for all vertices within the abstract word* ω*.*

(5.11)
$$\begin{aligned} \mathcal{G} : \mathcal{B}_V &\to \mathbb{B} \\ p &\mapsto \mathcal{G}^\omega p =_{def} (\forall v \in V : p(v)) \end{aligned}$$

5.3 Running Example

At the beginning of this section we have given some examples as a motivation for the extensions of the Edge Algebra. Now we want to give the formalization of the conditions:

- A network must have a signature:
  $\mathcal{G}self \Subset Network \Rightarrow (sig \Subset Signature \wedge |sig| = 1)$
- Ports of a signature must have a name:
  $\mathcal{G}(self \Subset Port \wedge (in \uplus out)^{-1} \Subset Signature) \Rightarrow (name \Subset String \wedge |name| = 1)$

- The vertices with the concept $Mult$ and Add must have at least two input ports and exactly one output port:
  $\mathcal{G}self \Subset Mult \uplus Add \Rightarrow (in \uplus out \Subset Port \wedge |in| \geq 2 \wedge |out| = 1)$
- Finally we want to formulate a corrected version for *dependsOn* introduced in Section 5.1:
  $dependsOn = simpleDependsOn \downarrow \sigma(\neg self \Subset Pre)$
  When replacing *simpleDependsOn* by *dependsOn* then
  $executionOrder(net) = \left\{ \begin{array}{ccc} \text{dt} & \rightarrow & \text{mult} \\ & & \searrow \\ & \text{pre} & \rightarrow \text{add} \end{array} \right\}$.

6 Abstract Languages

Up to now we have introduced abstract alphabets, abstract words and how to describe consistency conditions. In this section we finally want to define abstract languages according to the term *language* in formal languages and how we can use the Edge Algebra to describe abstract languages.

Definition 18 (Abstract language). *The set of all abstract words over an abstract alphabet $\Sigma = \langle C, P \rangle$ is denoted by Σ^*. An abstract language $\mathcal{L}$ over an abstract alphabet $\langle C, P \rangle$ is defined as a set of valid abstract words over Σ, thus $\mathcal{L} \subseteq \Sigma^*$.*

Since the set of valid abstract words is infinite in most cases, an explicit enumeration of all valid abstract words is insufficient. Instead of that abstract syntaxes are used to specify abstract languages. In general there are many ways of how to specify an abstract language. A very basic one is using the Edge Algebra:

Definition 19 (Abstract syntax based on the Edge Algebra). *An abstract syntax $\mathcal{S}$ is a tuple $\langle \Sigma, p \rangle$ where Σ is an abstract alphabet and $p \in \mathcal{B}_V$ is a node predicate.*

Then, the specified abstract language results to $\mathcal{L}(\mathcal{S}) =_{def} \{\omega \in \Sigma^* \mid \mathcal{G}^\omega p\}$.

This last definition finally allows us to specify abstract languages in a formal way without introducing a special metamodeling language. In order to formalize or compare various metamodeling approaches a mapping can be defined that translates dedicated metamodels into this formalism.

7 Conclusion and Future Work

In this paper we try to bridge the gap between formal and appropriate metamodeling approaches by adopting approved concepts and terms of formal languages. By using pomsets instead of sets we are able to deal with order and duplicates in a sound way. Additionally we have shown how to model enumerations and attributes like natural numbers or strings without introducing a special construct.

Decoupling the concrete metamodeling formalism from models allows a better re-use of the given terms and definitions for different metamodeling approaches. This paper should also provide a framework for formalizing or comparing different metamodeling approaches. The edge algebra allows the definition of inferred properties as well as complex consistency conditions.

A lot of open questions are not addressed within this paper. Instead of that this paper should form a basis for ongoing discussions in the metamodeling domain. Such topics are analysis of decidability and complexity of common problems like the word problem or equality and emptiness of languages. In this regard we can also ask about a language hierarchy like the Chomsky Hierarchy. Another interesting question is how to find a minimal word for a language if it exists. Regarding the Pomset operators and those of the Edge Algebra a set of theorems for restructuring formulars are still missing. In the context of databases containing a large abstract word this is also important for a query optimization. It also has to be verified whether the Edge Algebra is suitable for such an optimization. In addition a type system for the edge algebra would be helpful.

Based on these theoretical questions, also more methodical aspects are of interest: Important substitutes are how we can improve the quality of metamodels, what are metrics for the quality of metamodels, what constructs are necessary to create metamodels with a high quality, and how we can decompose a language definition into language modules. In order to give well-founded answers to all of these questions this paper provides a common formal basis.

References

1. Jouault, F., Bzivin, J.: Km3: a dsl for metamodel specification. In: Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037, Bologna, Italy (2006) 171–185
2. OMG: Meta object facility (mof) core specification 2.0. OMG Document formal/06-01-01 (1 2006)
3. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F., Cowan, J.: Extensible markup language (xml) 1.1 (second edition). Technical report, W3C (9 2006)
4. Bezivin, J., Kurtev, I.: Model-based technology integration with the technical space concept. In: Metainformatics Symposium 2005, Esbjerg, Denmark (2005)
5. Cattell, R., Barry, D., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., Velez, F.: The Object Data Standard: ODMG 3.0. Number ISBN: 1-55860-647-4. Morgan Kaufmann Publishers (1999)
6. Fallside, D.C., Walmsley, P.: Xml schema part 0: Primer second edition. Technical report, W3C (10 2004)
7. ISO: Extended ebnf (1996)
8. OMG: Unified modeling language: Superstructure. OMG Document formal/07-02-05.pdf (2 2007)
9. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6) (1970) 377–387
10. Rozenberg, G., ed.: Handbook of graph grammars and computing by graph transformation: volume I. foundations. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1997)

11. Nardi, D., Brachman, R.J., Baader, F., Nutt, W.: The description logic handbook: theory, implementation, and applications. Cambridge University Press, New York, NY, USA (2003)
12. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The generic modeling environment. In: Workshop on Intelligent Signal Processing, Budapest, Hungary. Volume 17. (May 2001)
13. OMG: Object constraint language 2.0 specification. OMG Document formal/06-05-01 (05 2006)
14. ISO: The standard query language (sql:2008) (2008)
15. Gottlob, G., Kappel, G., Schrefl, M.: Semantics of object-oriented data models - the evolving algebra approach. In: East/West Database Workshop. (1990) 144–160
16. Dar, S., Agrawal, R.: Extending sql with generalized transitive closure. IEEE Trans. on Knowl. and Data Eng. **5**(5) (1993) 799–812
17. Grumbach, S., Milo, T.: An algebra for pomsets. In: ICDT '95: Proceedings of the 5th International Conference on Database Theory, London, UK, Springer-Verlag (1995) 191–207
18. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. Number ISBN 978-0131425422 in Eclipse. Prentice Hall International (08 2003) EMF.
19. Poernomo, I.: A type theoretic framework for formal metamodelling. In: Architecting Systems with Trustworthy Components. Volume 3938/2006 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2006) 262–298
20. Boronat, A., Meseguer, J.: An algebraic semantics for mof. In: FASE. (2008) 377–391
21. Biegl, C.: Multigraph: an architecture for model-integrated computing. In: ICECCS '95: Proceedings of the 1st International Conference on Engineering of Complex Computer Systems, Washington, DC, USA, IEEE Computer Society (1995) 361
22. Brucker, A.D., Wolff, B.: A proposal for a formal ocl semantics in isabelle/hol. In: TPHOLs '02: Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics, London, UK, Springer-Verlag (2002) 99–114
23. Kyas, M., Fecher, H., de Boer, F.S., Jacob, J., Hooman, J., van der Zwaag, M., Arons, T., Kugler, H.: Formalizing uml models and ocl constraints in pvs. Electronic Notes in Theoretical Computer Science **115** (2005) 39 – 47 Proceedings of the Second Workshop on Semantic Foundations of Engineering Design Languages (SFEDL 2004).
24. Markovi, S., Baar, T.: An OCL semantics specified with QVT. In: Model Driven Engineering Languages and Systems. Volume 4199/2006 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2006) 661–675
25. Pratt, V.R.: The pomset model of parallel processes: Unifying the temporal and the spatial. In: Seminar on Concurrency, Carnegie-Mellon University, London, UK, Springer-Verlag (1985) 180–196
26. Kugele, S., Tautschnig, M., Bauer, A., Schallhart, C., Merenda, S., Haberl, W., Khnel, C., Mller, F., Wang, Z., Wild, D., Rittmann, S., Wechs, M.: Cola - the component language. Technical Report TUM-I0714, Technischen Universitt Mnchen (09 2007)
27. Merenda, S., Merenda, C.: Oomega - framework for model-based software engineering. http://www.oomega.net/
28. Eclipse-Foundation: Eclipse development platform. http://www.eclipse.org/