Enriching Megamodel Management with Collection-Based Operators

Rick Salay, * Sahar Kokaly, † Alessio Di Sandro, * and Marsha Chechik *
* University of Toronto, Canada
{rsalay, adisandro, chechik}@cs.toronto.edu
† McMaster University, Canada
kokalys@mcmaster.ca

***Abstract*—*Megamodels* are often used in MDE to describe collections of models and relationships between them. Typical collection-based operations – *map*, *reduce*, *filter* – cannot be applied directly to megamodels since these operators need to take relationships between models into consideration. In this paper, we propose adapted versions of these operators, demonstrating them on four megamodeling scenarios. We then analyze their applicability for handling industrial-sized megamodels. Finally, we report on a reference implementation of the operators and experimental results using it.**

I. Introduction

The use of models can benefit software engineering practice; however, a proliferation of models creates accidental complexity that must be managed. The field of Model Management [1] has emerged to address this challenge. Model management focuses on a high-level view in which entire models and their relationships (i.e., mappings between models) can be manipulated using specialized operators to achieve useful outcomes. For example, a model *match* operator [1] finds correspondences between the elements of two models and packages these as a mapping between the models. A *merge* operator [1] can then be used to combine the content of the two models using the correspondence information in the mapping. Model management approaches typically use *megamodels* [2] to represent sets of models and their relationships in this high-level view. For example, a megamodel could be a graphical model that uses nodes to represent models and edges to represent relationships.

Model management has been studied from many perspectives including algebraic properties of operators [3], [4], categorical foundations [5], type theory [6], megamodeling languages [7], [8] and practical implementations [9], [10], [4], [11]. In these investigations, the focus is on the general manipulation of models rather than specifically on the manipulation of megamodels – since these are kind of models, general model operators apply to them equally well. Yet other operators are needed due to the special role of megamodels in model management. Specifically, megamodels function as *collections* (of models and relationships) and so their manipulation should be like that of other collection types (e.g., lists, graphs, etc.) commonly found in modern programming languages. In particular, three collection operators are widely used: *map* for applying a function to every element of a collection, *reduce* for aggregating elements in a collection and *filter* for extracting a subset of the collection using a property as a selector. These operators would have great utility if available in the model management context. But while megamodels bear similarity to collections in programming, they also have their unique challenges that limit our ability to apply these techniques without some adaptation. We illustrate these below.

A Motivating Scenario. A company uses a megamodel to track its modeling artifacts (models and relationships between them). The company identified a particular construct of some of its models as undesirable (e.g., multiple inheritance in class diagrams), and now (1) would like to identify all models that are of type class diagram and contain this construct, (2) refactor them using a predefined transformation to remove the construct, and (3) merge the modified class diagrams in order to compare the result to the merged version of the original bad class diagrams. A natural way to execute these steps is to (1) use *filter* to extract the bad class diagrams *and the relationships between these*, (2) use *map* to apply the refactoring transformation to these and *also allow the use of the corresponding refactoring transformation for the relationships* and, (3) use *reduce* with a merge transformation to combine all the refactored models pairwise, *correctly taking into account the relationships between them*. The *reduce* with merge also can be used to combine the original models. Thus, we need collection operators to manipulate the entire *graphs* of related models rather than just lists of models. Furthermore, we need to allow invoking *map* and *reduce* with transformations that can accept graphs of models and relationships as input and produce these as output.

Contributions. We make the following contributions:
(1) We define the set of megamodel collection operators which treat relationships between models as first class entities:
- **map** – for applying a transformation to the elements of a megamodel;
- **reduce** – for aggregating the elements of a megamodel using a transformation; and
- **filter** – for extracting a subset of elements of a megamodel that satisfy a property.

(2) We analyze the complexity of the operators and discuss their applicability to industrial scale contexts.

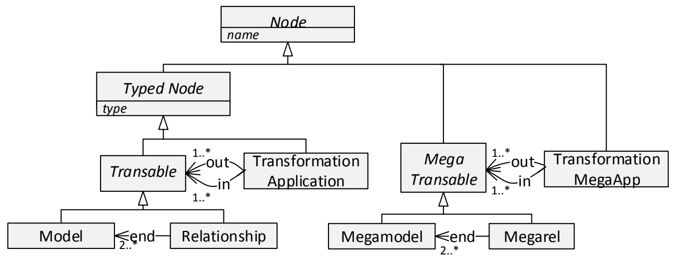

Fig. 1. Metamodel of an mgraph.

(3) We demonstrate the approach by using them to express several non-trivial megamodel management scenarios.
(4) We report on an implementation of the operators.

The rest of this paper is organized as follows: After fixing the terminology in Sec. II, we define the three collection operators for megamodels in Sec. III. Sec. IV illustrates these on four practical scenarios. Sec. V analyzes the complexity of the operators. Sec. VI describes tool support. We compare our approach with related work in Sec. VII and conclude in Sec. VIII with summary of the paper and a discussion of future research directions.

II. Preliminaries

A modeling environment typically consists of various modeling artifacts stored in *a repository*. A *megamodel* is a model whose elements refer to artifacts in the repository. In this section, we formalize the concept of megamodel and give other necessary definitions.

Basic Types. We define the concept of "mega-graphs" – *mgraphs*. A megamodel is an mgraph whose nodes refer to artifacts in the repository.

Definition 1 (mgraph): An *mgraph* is a structure that is an instance of the metamodel in Fig. 1. Given an mgraph G, we write G_C to denote the set of nodes in node class C. When C is omitted, the node class is Node. We use abbreviations Mod and Rel for node classes Model and Relationship, respectively. For $n \in G$, we write $n.R$ to denote the set of nodes on the other end of reference R from node n.

In this paper, we limit our focus to megamodels that can refer to artifacts corresponding to the concrete node classes in Fig. 1. A *relationship* is a mapping between two or more models on its ends. A *transformation application* is the record of having performed a given transformation on a set of input models and relationships to produce a set of output models and relationships. We make no further assumptions about the way models, relationships or transformation applications are represented or what they contain. The "mega" versions of these artifacts: megamodels, megarels and megaApps are defined below. We assume the existence of a repository.

Definition 2 (Repository): A *repository* $\mathscr{R}$ is a store for artifacts that is itself structured as an mgraph of artifacts (i.e., rather than an mgraph of symbols). Thus, a relationship has references to the models on its ends, etc.

Models, relationships and transformation applications are typed by model types, relationship types and transformations, respectively. We assume that type compatibility (e.g., via subtyping) is given by a relation TypeComp.

Definition 3 (Type compatibility): Given a type compatibility relation TypeComp, $\mathtt{TypeComp}(T, T')$ indicates that an artifact of type T can be used wherever the type T' is required. The relation TypeComp must be reflexive.

Mappings between mgraphs are called mgraph homomorphisms.

Definition 4 (mgraph homomorphism): Given mgraphs G, G' and type compatibility relation TypeComp, an *mgraph homomorphism* $f : G \to G'$ is a function $f_{\mathtt{Node}} : G_{\mathtt{Node}} \to G'_{\mathtt{Node}}$ that satisfies the following conditions for preserving all node classes C, references R and types:

1) $\forall n \in G \cdot n \in G_C \Rightarrow f_{\mathtt{Node}}(n) \in G'_C$
2) $\forall n, n' \in G \cdot n' \in n.R \Rightarrow f_{\mathtt{Node}}(n') \in f_{\mathtt{Node}}(n).R$
3) $\forall n \in G_{\mathtt{TypedNode}} \cdot \mathtt{TypeComp}(n.\mathtt{type}, f_{\mathtt{Node}}(n).\mathtt{type})$

A *typed mgraph homomorphism* is one where TypeComp is equality. An *mgraph isomorphism* is one where $f_{\mathtt{Node}}$ is a bijection.

Condition (1) ensures that f preserves node classes and condition (2) ensures that f preserves the endpoints of references. These are standard conditions for a homomorphism to be a structure-preserving mapping. Condition (3) additionally ensures that for typed nodes, f preserves type compatibility of nodes. Note that, node names need not be preserved by f.

Mega Types. Intuitively, all the mega types represent collections of artifacts.

Definition 5 (Megamodels): Let a model repository $\mathscr{R}$ of artifacts be given. A *megamodel* is a pair $\langle G, d\rangle$, where G is an mgraph and $d : G \to \mathscr{R}$ is a typed mgraph homomorphism, called the *dereferencing mapping*, that maps the nodes of G to the artifacts they represent in $\mathscr{R}$.

When it is clear from the context, we will use a megamodel interchangeably with its mgraph.

Definition 6 (Megarel): A *megarel* is a megamodel restricted to containing only Relationship and Megarel nodes but end references of these nodes are contained within the ends of the megarel.

Thus, a megarel is a "relationship-like" collection that itself has megamodels on its ends.

Definition 7 (megaApp): A *megaApp* is a megamodel restricted to containing only Transformation Application and Transformation MegaApp nodes but the in and out references of these elements are contained within the input and output connections of the megaApp. That is, both megarel and megaApp artifacts are connected to other artifacts in $\mathscr{R}$

Fig. 2 gives an example of a repository including three megamodels, a megarel (as well as other artifacts shown as shaded boxes). To avoid visual clutter in this example, the dereferencing mappings are not shown but are implied by the names; however, in general, names across the mapping may be different. The examples of the megamodels and megarels show the concrete syntax we use for illustrations in this paper. Models are shown as boxes, relationships are shown as diamonds with binary relationships shown optionally as a

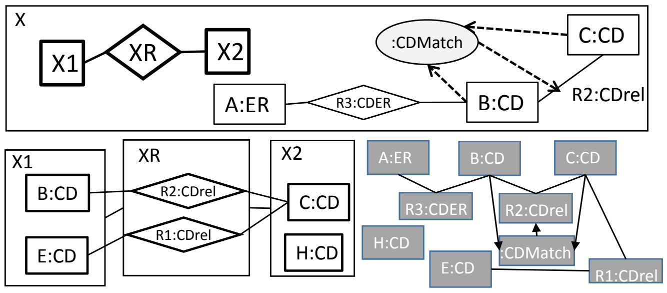

Fig. 2. An example of a repository including megamodels and a megarel showing the concrete syntax.

line. A transformation application is given as an oval, with the input elements connected with arrows pointing into the oval and output elements connected with arrows pointing out of the oval. All models, relationships and transformation applications have a label of form *name:type*, where the name is optional. Megamodels, megarels and megaApps are shown similarly as their non-"mega" counterparts but with thick borders. Furthermore, these elements are not typed. For example, in megamodel `X` at the top of the figure, the box with label `C : CD` refers to the class diagram with name `C`. The diamond `R2 : CDrel` refers to the corresponding `CRrel` artifact with name `R2`, the thick bordered box labeled `X1` refers to the megamodel `X1` shown below it which itself refers to models `B` and `E`, etc. No megaApp is shown but this will be illustrated in the subsequent sections of the paper.

Properties and Transformations. Models can satisfy properties and participate in transformations. We define these below.

Definition 8 (Property): A *property* is a constraint on an artifact. Given an artifact A and a property P, we write $A \models P$ to denote that A satisfies P. Every property is defined for an artifact of a specific type. If A has type T, P has type T' and the type compatibility relation `TypeComp` is given, the following condition must hold: $(A \models P) \Rightarrow \texttt{TypeComp}(T, T')$.

Thus, we assume that artifacts not compatible with the type of the property do not satisfy the property.

Definition 9 (Transformations): A *transformation* is a function that maps an mgraph of models and relationships to another mgraph of models and relationships. Given a transformation F, the *signature* of F is a pair $\langle I, O\rangle$ where $I \cup O$ is an mgraph, I is an mgraph called the *input signature* and O is a set of mgraph nodes called the *output signature*.

Note that we make no assumptions about what language is used for expressing properties or defining transformations.

Fig. 3 gives an example of the signature for a transformation `CDMerge` that accepts two class diagrams and a relationship between them and produces the merged class diagram with relationships back to the original two class diagrams. The input signature consists of the models a, b and relationship r and the output signature has model ab with relationships ra and rb. Written textually, the signature consists of $I = \{\mathtt{a} : \mathtt{CD}, \mathtt{b} : \mathtt{CD}, \mathtt{r}(\mathtt{a}, \mathtt{b}) : \mathtt{CDrel}\}$, and $O = \{\mathtt{ab} : \mathtt{CD}, \mathtt{ra}(\mathtt{a}, \mathtt{ab}) : \mathtt{CDrel}, \mathtt{rb}(\mathtt{b}, \mathtt{ab}) : \mathtt{CDrel}\}$. Even though a relationship is an output of a transformation, it can connect

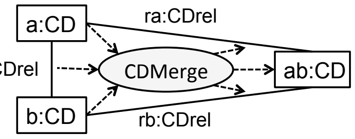

Fig. 3. Signature of a transformation `CDMerge` for merging class diagrams.

input elements. For example, both output relationships ra and rb are connected to inputs.

Definition 10 (Transformation binding): Given transformation F with signature $\langle I, O\rangle$, a *binding* K of F is a megamodel $\langle I, d_I\rangle$ where d_I is an mgraph homomorphism (rather than a *typed* mgraph homomorphism). We write $F(K)$ to denote the corresponding megamodel $\langle I \cup O, d_{I\cup O}\rangle$ giving the result of applying F to K. We say that F is *commutative* if for every pair of isomorphic bindings K, K' (i.e., mgraph isomorphisms of I), $F(K)$ is isomorphic to $F(K')$.

Thus, a binding K of F assigns artifacts to the nodes of its input signature I and then $F(K)$ can be evaluated to assign newly created artifacts to the nodes of the output signature. `CDMerge` in Fig. 3 is an example of a commutative transformation – given any two class diagrams `M1`, `M2` related by a relationship `R`, the merged output is the same regardless of whether we use the binding $\{\mathtt{a} := \mathtt{M1}, \mathtt{b} := \mathtt{M2}, \mathtt{r} := \mathtt{R}\}$ or $\{\mathtt{a} := \mathtt{M2}, \mathtt{b} := \mathtt{M1}, \mathtt{r} := \mathtt{R}\}$.

Definition 11 (binding with megamodel): Given megamodel $X = \langle G_X, d_X\rangle$ and transformation F with signature $\langle I, O\rangle$, a *binding of* F *within* X is the megamodel $\langle I, d_X \circ b\rangle$ where b is an injective mgraph homomorphism $b : I \to X$.

That is, a binding of F within X is formed by finding a set of nodes in X that match I.

Traditional Megamodeling Operators. As discussed in Sec. I, a number of model management operators have been defined, with *match*, *merge*, *diff*, and *slice* among them. For the illustrations in this paper, we require only one of them – a simple type of megamodel merge that we call **union**. We define it in more detail below. The **union** operator combines the content of a set of megamodels into a single megamodel in which elements that refer to the same artifact are merged into a single element.

There are two possibilities for the set of input megamodels: (1) either they are an mgraph of megamodels and megarels, or (2) they are a set of megarels that share the same endpoints. Fig. 4 illustrates both cases. In case (1), the result is a megamodel while in case (2) it is a megarel with the same endpoints as the inputs.

The union process can cause conflicts coming from the following two sources. If megamodel elements refer to the same artifact but the names of these elements differ, it is not clear which name to use for the merged element. To resolve this, we assume that the names in the union is a combination of the original names. Another conflict occurs when different artifacts are referred to by different elements using the same name. In this case, we assume that the names are made distinct in the union. Both of these conflict scenarios are illustrated at the bottom of case (1) in Fig. 4. Both `A` and `D` refer to the

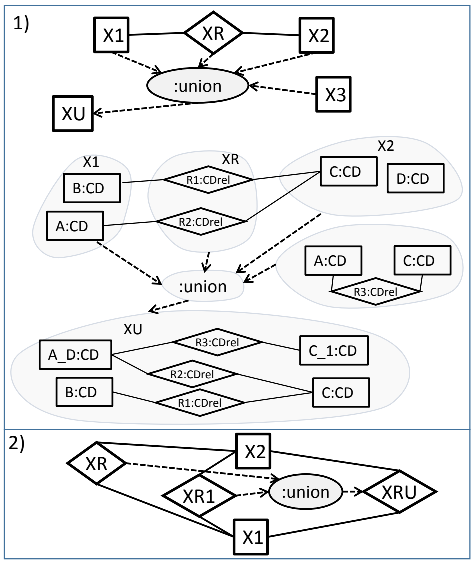

Fig. 4. An illustration of the **union** operator applied to (1) an mgraph of megamodels (megamodel contents shown underneath), and (2) a set of megarels that share the same endpoints.

same model and in the union the name `A_D` is used. However, the element `C` in `X2` refers to a different model than `C` in `X3` and the latter is assigned the name `C_1` in the union.

III. Megamodel Collection Operators

In this section, we define the set of megamodel collection operators we are proposing in this paper: **map** – Sec. III-A, **reduce** – Sec. III-B and **filter** – Sec. III-C. Their signatures are $\mathbf{map}[\mathcal{T}] : \mathscr{P}(\mathcal{M}) \rightarrow \mathscr{P}(\mathcal{M})$, $\mathbf{reduce}[\mathcal{T}] : \mathcal{M} \rightarrow \mathcal{M}$, and $\mathbf{filter}[\mathcal{P}] : \mathcal{M} \rightarrow \mathcal{M}$, respectively, where $\mathcal{T}$ is the set of model transformations, $\mathcal{M}$ is the set of megamodels, $\mathcal{P}$ is the set of model properties and $\mathscr{P}$ is the powerset operator. All three are higher-order operators that accept a transformation or a model property as a parameter (indicated in square brackets). We describe each operator as follows: first the standard usage, then the special adaptation needed to handle megamodels and finally, the behaviour defined as an algorithm.

A. Operator map

Background: Standard Usage. The usual behaviour of a **map** operation is to traverse a collection (e.g., list, tree, etc.) and apply a function to the value at each node in the collection. The result is a collection with the same size and structure as the original with the function output value at each node. For example, given the list of integers $L = [10, 13, 4, 5]$ and the function $Double$ that takes an integer and doubles it, applying map with $Double$ to L yields the list $[20, 26, 8, 10]$. If the function has more than one argument, the mapped version can take a collection (with the same size and structure) for each argument, and the function is applied at a given node in the collection using the value at that node in each argument in the collection.

Adaptation for Megamodels. Since a transformation input signature is an mgraph, applying it to each node of a megamodel is not possible. Instead, the **map** operator for megamodels applies the transformation for every possible binding of the input signature in the input megamodel(s). The collection of outputs from these applications forms the output megamodel.

When the transformation signature consists of a single input and output type and uses a single input megamodel which happens to be a set (i.e., no relationships) of instances of the input type, then our **map** produces the same result as a standard map operator applied to a set. However, in the general case, **map** is more complex and differs from the behaviour of the standard map. In particular,

(1) The output megamodel may not have the same structure as the input megamodel since the structure is dependent on the output signature of the transformation.

(2) The size of the output may not be equal to the size of the input. For example, if a transformation `FF` takes two models as input and produces one as its output, applying **map** to it on a megamodel with n models will produce as many as $n \times (n-1)$ output models since each pair of input models may be matched in a binding. At the other extreme, if no input models form a binding then the output will be the empty megamodel.

(3) When there are multiple input megamodels, each binding of the input signature is split across the input megamodels in a user-definable way.

(4) When the transformation is commutative, we (may) want to avoid replication in the output due to isomorphic bindings. For example, if the transformation `FF` is commutative, we will get each output model twice since there are two ways to apply `FF` to a pair of models.

In what follows, we propose an operator **map** for handling megamodels while avoiding the above problems.

Definition. $\mathbf{map}[F](\{X_e | e \in I\})$ applies a model transformation F with a signature $\langle I, O\rangle$ to a set of input megamodels $\{X_e | e \in I\}$. Note that the megamodels X_e need not be distinct; thus multiple input arguments can be taken from the same megamodel. **map** produces an output megamodel for each element of the output signature O. The behaviour is defined by the algorithm in Fig. 7.

We explain the algorithm using the illustration in Fig. 6(1) of applying **map** to the `CDMatch` transformation given in Fig. 5. The input signature consists of $\{\mathtt{a} : \mathtt{CD}, \mathtt{b} : \mathtt{CD}\}$ and the output signature is $\{\mathtt{r}(\mathtt{a}, \mathtt{b}) : \mathtt{CDrel}\}$. The diagram at the top of Fig. 6 shows $\mathbf{map}(\mathtt{CDMatch})$ applied to megamodels `X1` and `X2` to produce an output megarel `XR`. Thus, the input megamodels are $X_a := \mathtt{X1}$ and $X_b := \mathtt{X2}$ and the one output, Y_r, corresponding to the output signature element `r`, produces the value for `XR` .

In line 1, the output megamodels are initialized to the empty megamodel. In our example, $Y_r = \emptyset$. Lines 2-5 iterate over all possible bindings of I in the input megamodels. In line 2, a fresh binding (i.e., previously unmatched) for the input signature of F is found in the input megamodels. Thus, in this example, a binding for `a` is drawn from `X1` and a binding for `b` – from `X2`. Assume this is $K := \{K_a := \mathtt{A}, K_b := \mathtt{C}\}$. Lines 3-4 check whether isomorphic bindings should be ignored because F is commutative. Binding isomorphisms do not

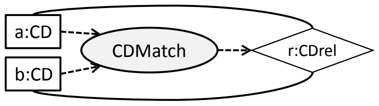

Fig. 5. Signature of the `CDMatch` transformation.

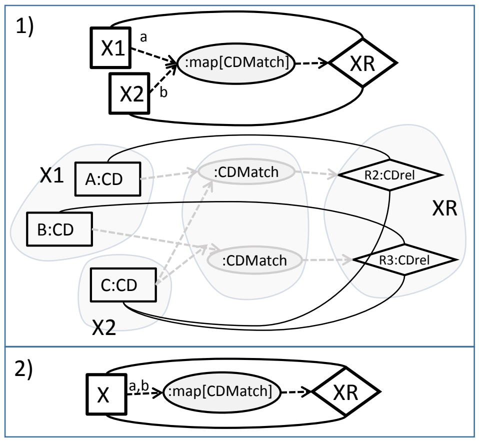

Fig. 6. 1) An illustration of applying **map** to the `CDMatch` transformation using two input megamodels (megamodel content shown underneath); 2) using the same input megamodel for both arguments.

occur in this example, so we illustrate them separately below. In line 5, the output of applying the transformation to the combined input binding is added to the output megamodels. Thus, in our example, `CDMatch` is applied to K and the resulting `CDrel` relationship `R2` is added to Y_r. Line 6 returns the resulting output.

In our example, there are only two matches; thus, the resulting megarel contains two relationships. However, consider the alternative application of **map** to `CDMatch` shown in Fig. 6(2). Here both input elements are taken from the input megamodel `X`. Assume that `X` contains all three models $\{\mathtt{A:CD}, \mathtt{B:CD}, \mathtt{C:CD}\}$. In that case, there are six possible ways to match the input signature. However, since `CDMatch` is designated as `commutative`, a binding $\{K_a := m, K_b := n\}$ produces the same output as $\{K_a := n, K_b := m\}$; thus, only three matches are used to produce the output.

B. *Operator* reduce

Background: Standard Usage. There are different variants of the reduce (also called fold, aggregate, etc.) operator used in programming languages but it typically accepts a binary function F and applies it over values $x_1, x_2, \ldots, x_n$ in a recursive collection (e.g., list, tree, etc.) by accumulating the intermediate values, e.g., $F(x_n, F(\ldots, F(x_3, F(x_2, x_1))$. For example, applying reduce with the "+" operator to the list $[1, 3, 1, 9]$ produces the sum 14.

Adaptation for Megamodels. In a similar way, we expect the **reduce** operator to accept a transformation F and use this to combine the elements of the input megamodel. Our approach is to view F as a rewrite rule, by repeatedly applying F in-place and deleting the input elements until it can no longer be applied. First, we must consider several issues:

```
Algorithm: Apply map
Input: transformation $F$ with signature $\langle I, O \rangle$,
       megamodels $\{X_e | e \in I\}$
Output: set of megamodels $\{Y_e | e \in O\}$
1:  for $(e \in O)$ { let $Y_e := \emptyset$ }
2:  for (fresh binding $K$ in $\{X_e | e \in I\}$) {
3:      if $F$ is commutative then
4:          if isomorphism of $K$ already done then continue;
5:      for $(e \in O)$ { add element $e$ of $F(K)$ to $Y_e$ } }
6:  return $\{Y_e | e \in O\}$
```

Fig. 7. Algorithm defining behaviour of the **map** operator.

(1) What should be the criteria that F must satisfy for this process to terminate?

(2) Since a megamodel is not a recursively defined structure and has no well-defined ordering on its elements, we cannot rely on a specific traversal path. Thus, F must be *confluent* – the final result of **reduce** should be same regardless of the order in which we apply F to the megamodel.

(3) Since the input elements may have relationships to other neighbouring elements in the megamodel, we must be careful to preserve this information when the relationships are deleted.

We will address issues (1) and (2) in the definition of **reduce** below with appropriate assumptions on F. We address issue (3) by using relationship composition operators to construct new relationships to neighbouring elements as needed. As an illustration, assume we are using **reduce** with the `CDMerge` transformation (See Fig. 3) to merge a megamodel of class diagrams and `CDRel` relationships. Fig. 9 shows one iteration of the reduction. In step ❶, `CDMerge` is applied to an arbitrarily chosen pair of models (in this case, `B` and `C`) to produce a new class diagram `BC`. In step ❷, composition operators are invoked to connect `BC` to the neighbours of `B` and `C`. Finally, in step ❸, the original models `B` and `C` are deleted together with all of their relationships.

Definition. We now define a new operator **reduce**$[F](X)$ aimed to apply a transformation F to reduce the content of a megamodel X. We begin by making the following assumptions: (I) We assume availability of predefined relationship composition operators for all relationship combinations we encounter, together with a library function `getCompOp` that provides such an operator given a pair of relationship types.

(II) In order to achieve confluence, F is required to be commutative and associative with itself and with all relationship composition operators used in item (I).

(III) In order for the reduction process to terminate, we put the constraint on F that it must be strictly reducing in output types: for every model type in the input signature, there must be fewer models of that type in the output signature; and, for relationship type in the input signature, there must be fewer relationships of that type in the output signature that are connected to output models on both (or all, for n-ary) ends.

Fig. 8 gives the algorithm for defining the behaviour of **reduce**. In line 1, Y is initialized to the same value as the input. Lines 2-9 iterate for each binding of F in Y until no

Algorithm: Apply reduce
Input: transformation F with signature $\langle I, O \rangle$,
megamodel X
Output: megamodel Y

```
1:  let $Y := X$
2:  for (binding $K$ in $Y$) {
3:      apply $F(K)$ generating output $K'$;
4:      for ($m \in K_{\texttt{Mod}}, m' \in K'_{\texttt{Mod}}, r(m, m') \in K'_{\texttt{Rel}}$) {
5:          for ($m'' \in Y_{\texttt{Mod}}, r'(m'', m) \in Y_{\texttt{Rel}}$) {
6:              let $comp := \texttt{getCompOp}(\texttt{type}(r'), \texttt{type}(r))$;
7:              let $r''(m', m'') := comp(r', r)$;
8:              add $r''$ to $Y$ }}
9:      delete elements in $K$ from $Y$ }
10: return $Y$
```

Fig. 8. Algorithm defining behaviour of the **reduce** operator.

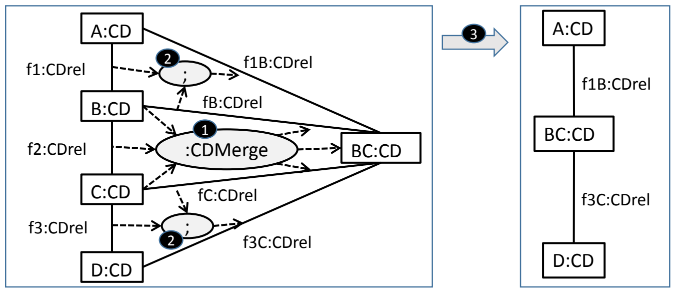

Fig. 9. An illustration of one iteration of **reduce**. First the merge is applied non-deterministically (step 1). Then the relationships to the neighbours of the merged models are computed using appropriate composition operators. Finally, all input elements are deleted.

more can be found and the algorithm terminates returning Y (line 10). In the loop, for a given binding K (line 2), F is first applied to get K' line 3. Then lines 4-9 perform the steps as described in Fig. 9 to connect the neighbours of input models in K to the output models in K' using composition operators and then deleting the input models in K. For each output model m' with a relationship r to an input model m (line 4), and for each neighbour model m'' of input model m with relationship r' (line 5), a new relationship r'' is constructed directly from m'' to m' by composing r' and r (line 7). The operator to compose a relationship of $\texttt{type}(r')$ with one of $\texttt{type}(r)$ must be "looked up" using `getCompOp` (line 6).

C. *Operator* filter

Background: Standard Usage. Many languages provide a filtering operation to extract a portion of collection that satisfies some condition. For example, filtering the list $[2, 5, 6, 8, 9, 1]$ using the property `isEven` produces the list $[2, 6, 8]$.

Adaptation for Megamodels. The **filter** operator is similar and applies to megamodels. A property is given as the filtering condition, and the subset of elements that satisfy the property is used to produce the output. We distinguish between model and relationship properties and treat them independently. Thus, a *model property* filters only models and keeps all relationships

Algorithm: Apply filter
Input: property P, megamodel X
Output: megamodel Y

```
1:  let $Y := \emptyset$;
2:  for ($m \in X_{\texttt{Mod}}$) {
3:      if $P$ is a model property then
4:          if $m \models P$ then add $m$ to $Y$;
5:      else add $m$ to $Y$ }
6:  for ($r \in X_{\texttt{Rel}}$) {
7:      if $P$ is a relationship property then
8:          if $r \models P$ then add $r$ to $Y$;
9:      else if $r.\texttt{end} \cap Y \neq \emptyset$ then add $r$ to $Y$ }
10: return $Y$;
```

Fig. 10. Algorithm defining behaviour of the **filter** operator.

between the remaining models. A *relationship property* filters only relationships and does not affect the models.

filter differs from **map** and **reduce** in that it does not create new models or relationships; it just creates new references to existing models and relationships. Thus, all elements of the output megamodel refer to artifacts that are already referred to by elements of the input megamodel. This aspect of **filter** makes it an inexpensive operation compared with **map** or **reduce**.

If a property P, defined for a model or relationship type T, is used for **filter**, then it selects all elements of type T (or its compatible types) that satisfy the constraints in P (See Defn. 8). It is also possible to give a type T as the property which is interpreted as the property *true*, satisfied by any instance of T (or its compatible types).

Definition. **filter**$[P](X)$ filters megamodel X to produce the least sub-megamodel of X containing all the elements of X that satisfy property P.

The behaviour of **filter** is given by the algorithm in Fig. 10. Line 1 initializes the output to the empty megamodel. Lines 2-5, iterate over the model elements in X. If P is a model property then the model is only added to the output if passes the satisfaction check (line 4). If P is not a model property, all models are added to the output (line 5). A similar algorithm is followed in lines 6-9 that iterate over relationship elements. The only difference is that if P is not a relationship property (and so it must be model property), only those relationships are added to the output that already have their endpoints in the output due to the filtering in lines 2-5.

IV. Application Scenarios

In this section, we illustrate our collection-based megamodel management operators using several scenarios.

A. *Experiment Driver*

The goal of this scenario is to apply a transformation on a megamodel and perform some kind of experiment on the result of its application. Specifically, given a megamodel `XCD` containing a set of class diagrams, we wish to apply transformation `CD2Java` that translates a class diagram to its

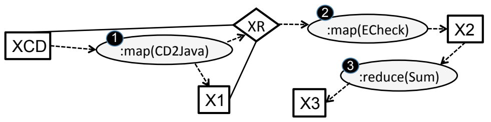

Fig. 11. Experiment driver scenario illustration.

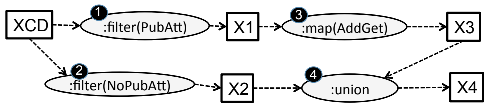

Fig. 12. Mass refactoring scenario illustration.

equivalent Java code and produces a CD2JavaRel traceability relationship from the CD to the Java code. Then, we wish to apply evaluation transformation ECheck on each CD2JavaRel in the megarel resulting from the transformation application. ECheck computes the number of classes in each transformed class diagram that do not have Java counterparts. Finally, we would like to sum these up via a Sum operation to learn the total number of incidents where this occurs. If this is greater than zero, then we will identify a problem in the transformation. Fig. 11 shows the chain of operators required to accomplish this via the following steps:

(1) Apply **map**[CD2Java](XCD) to produce X1 which contains the Java code and XR which is the megarel containing all relations between XCD and X1.

(2) Apply **map**[ECheck](XR) to produce megamodel X2 which contains the evaluation ECheck for each rel in XR.

(3) Apply **reduce**[Sum](X2) to produce the final result X3 containing a single value which is the sum of the results of **map**[ECheck](XR). A value is greater than zero indicates that there was a problem in the transformation application.

B. Mass Refactoring

We are given a megamodel XCD that contains unrelated class diagrams, a property PubAtt that represents models with public attributes and its negation NoPubAtt. We wish to find models satisfying PubAtt and refactor them so that public attributes become private attributes with public getter methods using refactoring transformation PubGet. Fig. 12 illustrates this scenario via the following steps:

(1) Apply **filter**[PubAtt](XCD) to produce a megamodel X1 containing the sub-megamodel of XCD with models where property PubAtt holds.

(2) Apply **filter**[NoPubAtt](XCD) to produce a megamodel X2 containing the sub-megamodel of XCD with models where property PubAtt does not hold.

(3) Apply **map**[AddGet](X1) to transform the models with the undesirable property using a refactoring transformation AddGet which produces a megamodel X3.

(4) Return a megamodel X4 = **union**(X2, X3) (see Sec. II) which represents the refactored version of the original s.t. the property PubAtt no longer holds on any of its models.

C. Megamodel Transformation

We are given an input megamodel XCD consisting of class diagrams (CDs) related by class diagram relations (CDrels), and

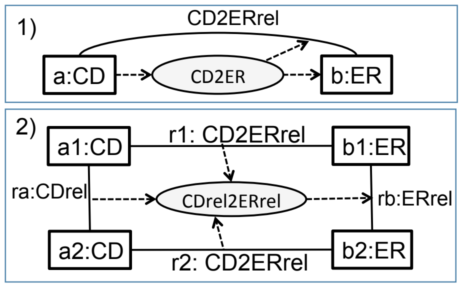

Fig. 13. Illustration of transformation signatures for megamodel transformation scenario. (1) Class Diagram (CD) to Entity Relationship (ER) transformation, (2) CD relation to ER relation transformation.

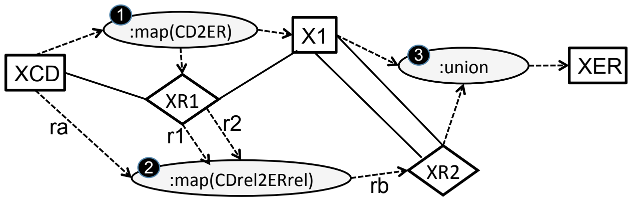

Fig. 14. Megamodel transformation scenario illustration.

we wish to transform it to a megamodel XER consisting of ER diagrams (ERs) related by ER diagram relations (ERrels). We are also given the transformations CD2ER and CDrel2ERrel (See signatures in Fig. 13) which transform CDs to ERs and CDrels to ERrels, respectively. We would like to use our operators to accomplish this.

The steps to perform this transformation are illustrated in Fig. 14 and involve the following steps:

(1) Apply **map**[CD2ER](XCD) which based on its signature applies only to the (CDs) in XCD and produces the megamodel X1 consisting of the ER versions of all the CDs in XCD as well as the megamodel relation XR1.

(2) Apply **map**[CDrel2ERrel](XCD) which based on its signature applies only to the CDrels in XCD and produces the megamodel relation XR2 consisting of a set of ERrels with endpoints in X1. Note that the other arguments come from the megamoodel relation XR1 which contains the applications of the CD2ER transformation.

(3) Apply **union**(X1, R) to produce the final megamodel XER which contains the corresponding ERs and the ERrels between them.

D. Motivating Example from Sec. I

Recall the motivating example in Sec. I: given a megamodel XCD which contains class diagrams and an undesirable property Mi that represents class diagrams with multiple inheritance, we aim to identify all models that are of type class diagram and contain this property, refactor them using a predefined transformation to remove the property, and merge the modified class diagrams. Fig. 15 shows the chain of operators required to accomplish this scenario:

(1) Apply **filter**[Mi](XCD)to produce a megamodel X1 containing the sub-megamodel of XCD with models where property Mi holds.

(2) Based on the megamodel transformation pattern described in Scenario C: apply **map**[RemoveMi](X1) to pro-

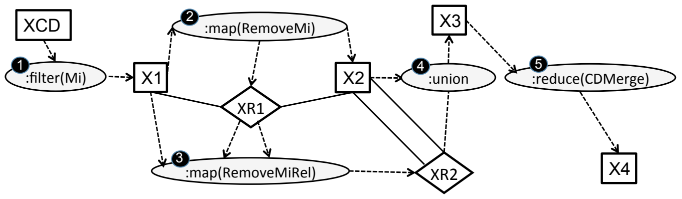

Fig. 15. Motivating example illustration.

map$[F](\{X\})$	$O(n^k \times C_F(m))$
reduce$[F](X)$	$O(n^2 \times C_F(m))$
filter$[P](X)$	$O(n^q \times C_P(m))$

Fig. 16. Worst case complexity of the operators.

duce X2 which is the refactored version of X1 that no longer contains the undesirable property, and (3) apply **map**[RemoveMiRel](XR1) to produce the megamodel XR2 containing the relations between the refactored models. (4) Apply **union**(X2, XR2) to produce X3 which is the megamodel containing the refactored models and relations between them.

(5) Apply **reduce**[CDMerge](X3) which applies the CDMerge operation described in Sec. III on class diagrams with relation CDRel between them and produces a megamodel X4 where all the related class diagrams are now combined. The final result can now be compared with the result of merging the pre-refactored models which can be achieved by using **reduce**[CDMerge](XCD).

Although the scenarios we have presented address specific types of megamodels, transformations and properties, they can be generalized as design patterns for similar reoccurring problems. For example, the mass refactoring scenario can be generalized for any problem that involves a megamodel which may contain elements with a certain property which should be removed. Similarly, the megamodel transformation scenario can be generalized for any problem that involves a transformation of one type of megamodel to another, given the appropriate transformations between source and target models and source and target relations. We have observed that in the case of the megamodel transformation pattern, the relation transformation can be induced from the model transformation; however, further analysis is outside the scope of this paper.

V. Analysis

In this section, we analyze the worst case complexity of the three operators we propose – see the summary in Fig. 16 – and discuss the implications of this for scalability.

Complexity of map. For the algorithm in Fig. 7, the iteration in line 2 over possible bindings of input signature I can execute up to n^k times, where n is the number of models input megamodels and k, the number of models in I. If F is commutative with q isomorphisms of I, the loop can execute $(n^k)/q$ times. Thus, the complexity is $O(n^k \times C_F(m))$ where $C_F(m)$ is the complexity of executing F in terms of a size metric m of the input binding to I. We assume that there is at most one relationship of each type between any given set of models in input megamodels.

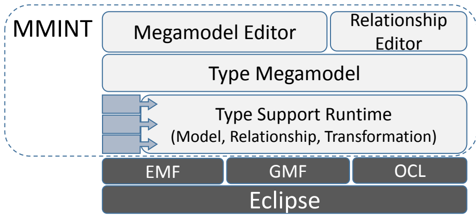

Fig. 17. Architecture of *MMINT* .

Complexity of reduce. Line 2 of the algorithm in Fig. 8 iterates over all possible bindings of input signature I, but each time, the input models and relationships are deleted. Thus, each input element participates in at most one binding. Furthermore, due to the assumption that F is strictly reducing, each iteration reduces the number of models and relationships. Thus, the number of iterations is bounded by n, the number of models in X – as with **map** we assume there is at most one relationship of each type between a given set of models. The internal loops lines 4-8 iterate once for every neighbour of a model M in I and relationship of M to a model in O. This can iterate up to rn times where r is the number of relationships between O and I. Since r is a constant for a given F, the complexity is $O(n^2 \times C_F(m))$.

Complexity of filter. For a property, the algorithm in Fig. 10 iterates n times while for a relationship property over a q-ary relationship, it iterates wn^q times, where n is the number of models in X and w is the number of q-ary relationship types. Since we assume w is a constant, the complexity is $O(n^q \times C_P(m))$ where $C_P(m)$ is the complexity of checking P in terms of a size metric m of the input relation.

Discussion. The analysis results in Fig. 16 show that the operators scale reasonably for certain classes of application scenarios. Specifically, the complexity is no worse than quadratic (modulo the transformation/property complexity) in the size of the input megamodel when **map** is applied to a transformation with two or fewer input models, in all cases for **reduce** and when **filter** is applied to either a model property or to a binary relationship property. Some scenarios exceed these limits (e.g., scenario C in Sec IV); we discuss future work for addressing scalability in Sec. VIII.

VI. Tool Support

A. MMINT *Overview*

The megamodel collection operators described in this paper has been implemented on the *MMINT* (Model Management INTeractive) workbench. *MMINT* is implemented[1] in Java and extends the MMTF model management framework [10]. *MMINT* uses the Eclipse Modeling Framework (EMF) [12] to express models and the Eclipse Graphical Modeling Framework (GMF) to create custom editors for editing models and relationships. The overall architecture of *MMINT* is illustrated in Fig. 17.

MMINT uses a distinguished *type megamodel* in which model types, relationship types and transformations are regis-

[1]Available at: http://github.com/adisandro/MMINT

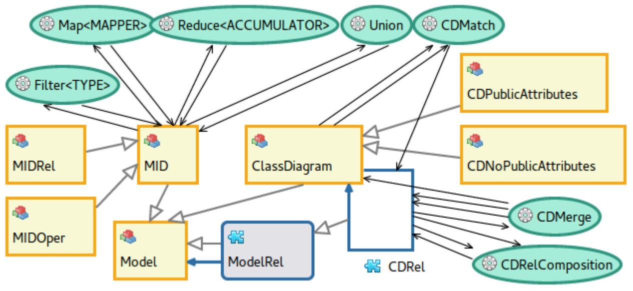

Fig. 18. Type megamodel in *MMINT* used for the examples in this paper.

tered. Fig. 18 shows a screenshot of the type megamodel used to implement examples in this paper. Here, boxes represent model types and links between them are binary relationship types (thick blue arrows). The sub-typing between types is shown with the hollow-headed arrows. Transformations are ovals connected to their input and output types with named links (names are not shown to avoid clutter). The transformation signature information can be extracted directly from this model. Additional metadata such as whether a transformation is commutative or is a relationship composition relation is also stored in this model.

The runtime operation of *MMINT* is centred around a megamodel editor that allows an engineer to interactively create models and relationships, invoke transformations on them and inspect the results. Implementations for supporting tools such as type-specific editors, validation checkers, solvers and custom transformation implementations can be plugged in and are managed by the type support runtime layer. A generic relationship editor is built into *MMINT* .

B. Implementation of Collection Operators

In *MMINT* a megamodel is referred to as a `MID` (Model Interconnection Diagram). All transformations, including higher-order ones, are registered in the type megamodel. Thus, the three collection megamodel operators in this paper can be seen at the top of Fig. 18 with their inputs and outputs connected to the `MID` type indicating that they take megamodels as input and produce them as output. In addition, each accepts a parameter (within the angle brackets). **union** can also be seen as an unparameterized transformation. The type compatibility relation (see `TypeComp` from Def. 3) is given by the subtype relation in the type megamodel.

Properties are implemented in *MMINT* as a model or relationship subtype that contains additional well-formedness constraints but does not change the metamodel of its supertype. For example, the `CDPublicAttributes` type is used for scenario B in Sec. IV and contains the following OCL code:

```
CDPublicAttributes:
  Attribute.allInstances()->exists(
    attribute | attribute.public)
```

The algorithms in Fig. 7, 8 and 10 for the three operators (and **union**) are implemented in Java and plugged into the type support runtime layer as transformation definitions. At runtime, when an engineer selects one or more MID elements and right-clicks to see what transformations are available to apply, they see these operators and can select one to apply. If it is parameterized, then a second dialog appears showing the choices for the parameter. For example, Fig. 19 shows a screenshot of scenario B in Sec. IV being built in the megamodel editor. Currently, the engineer is on step 3 and is invoking the **map** operator.

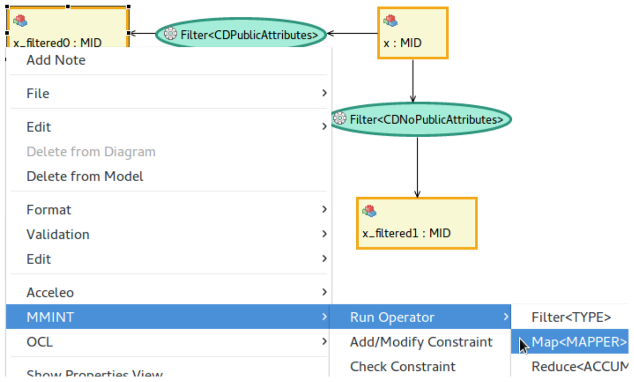

Fig. 19. Screenshot of megamodel for scenario B in Sec. IV being built in the *MMINT* megamodel editor.

TABLE I
EXPERIMENTAL RESULTS RUNNING **map**[`CDMatch`]

	# CDs	# rels	time (sec)	MID size (MB)
exp1	10	100	0.15	0.2
exp2	100	10000	12.92	20.7
exp3	250	62500	85.74	128.9
exp4	500	250000	422.78	518.7

C. Experiments

The *MMINT* implementation was used to express each of the four scenarios described in Sec. IV. Although these are "toy" experiments, they exercise the different aspects of the implementation. As a preliminary robustness test of the implementation, we ran an additional experiment in which we populated a megamodel with a varying number of class diagrams. The class diagrams were generated to contain 5 distinct random classes, picked from a pool of 50 available classes. Thus, many of the class diagrams share classes with the same name. We then measured the running time of **map**[`CDMatch`], given that **map** is the most complex of the operators, and the memory size of its output. The results are shown in Table I.

Since `CDMatch` has two input models, the complexity formula in Fig. 16 predicts quadratic time in the number of models of the input megamodel. The results in Table I are consistent with this prediction and additionally show that the space also increases quadratically. Although 422s ($\sim$7min) does not seem excessive to process 500 models, we plan to improve scalability further (See Sec. VIII).

VII. Related Work

Many model management approaches have been proposed. For example, Rondo [4] represents models as directed labeled graphs and supports traditional model management operations (e.g., match and merge) that work directly on models but not on the megamodels containing them. Maudeling[2] offers advanced query services; however, these are on the modeling artifacts themselves and not on megamodels. Epsilon [11] provides a set of domain specific languages for specific model management operations such as match and merge; however, no special support is provided for megamodels.

The Atlas Model Management Architecture (AMMA) [13] has a component AM3 for expressing megamodels and an OCL-based scripting language MoScript for general model management scripts including limited support for megamodel manipulation. Specifically, MoScript [9] provides support for *map* by using the OCL ApplyTo and Collect operations and support for *filter* using the OCL Select operation; however, these versions of map and reduce are more limited than what we propose because MoScript does not treat relationships between models as first class citizens and the support for map and reduce is limited to sets of models rather than graph-like collections in megamodels. In addition, MoScript does not provide support for the reduce operation. Despite these weaknesses, we see MDE workflow languages such as MoScript, UniTI [14], and TraCo [15] as complementary to our approach and believe they can benefit from incorporating our megamodel manipulation operations into the language. We leave the investigation of such integration for future work.

Model search engines such as MOOGLE [16] or IncQuery [17] perform queries of model contents. Our **filter** operation does not limit which languages or engines can be used for defining model and relationship properties. Thus, model search engines are complementary to our approach.

Graph-based languages and frameworks that provide collection-based operations on graphs have been proposed. The map and fold (i.e., reduce) algorithms in [18] generalize the classic list-based versions of these to graphs but the assumptions made by these algorithms make them inapplicable to the megamodel case. Specifically, the map algorithm does not allow for a "graph" of input arguments to the transformation as **map** does with transformation input signatures, and the fold algorithm only aggregates values on nodes and edges rather than collapsing the graph structure itself as **reduce** does. The MapReduce approaches of Google and others [19] are intended for the efficient processing of big data; yet these operate differently from the *map* and *reduce* functions found in many programming languages [20].

VIII. Conclusion and Future Work

In this paper, we have proposed three new megamodel collection operators: **map**, **reduce**, and **filter**. These operators are inspired by similar collection manipulation operators found in many programming languages, but are adapted to address the special characteristics of megamodels and MDE environments. Specifically, the operators treat model relationships as first class entities and address the graph-like structure of megamodels and of the signatures for model transformations.

[2]Maudeling:http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Maudeling

Our future work will explore several issues. First, since our operators are currently standalone, we want to investigate how best to integrate them into an MDE workflow language. This includes approaches for validating workflows that use our operators. Second, we will consider how to extend our operators to take into account megamodel hierarchical structure. Finally, since the combinatorial nature of **map** limits its scalability, we intend to investigate ways to mitigate this problem. For example, it may be possible to adapt the highly parallelizable MapReduce framework used in big data scenarios. Our overall objective in these investigations is to produce a set of scalable megamodel manipulation operators that are needed in typical model management scenarios.

References

[1] P. A. Bernstein, "Applying Model Management to Classical Meta Data Problems," in *Proc. of CIDR'03*, vol. 2003, 2003, pp. 209–220.

[2] J. Bézivin, F. Jouault, and P. Valduriez, "On the Need for Megamodels," in *Proc. of OOPSLA/GPCE Workshops*, 2004.

[3] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh, "A Manifesto for Model Merging," in *Proc. of GAMMA at ICSE'06*, 2006, pp. 5–12.

[4] S. Melnik, E. Rahm, and P. A. Bernstein, "Rondo: A Programming Platform for Generic Model Management," in *Proc. of SIGMOD'03*. ACM, 2003, pp. 193–204.

[5] Z. Diskin, S. Kokaly, and T. Maibaum, "Mapping-Aware Megamodeling: Design Patterns and Laws," in *Proc. of SLE'13*, 2013, pp. 322–343.

[6] A. Vignaga, F. Jouault, M. C. Bastarrica, and H. Brunelière, "Typing Artifacts in Megamodeling," *J. Software & Systems Modeling*, vol. 12, no. 1, pp. 105–119, 2013.

[7] R. Salay, J. Mylopoulos, and S. Easterbrook, "Using Macromodels to Manage Collections of Related Models," in *Proc. of CaiSE'09*. Springer, 2009, pp. 141–155.

[8] J.-M. Favre, R. Lämmel, and A. Varanovich, *Modeling the Linguistic Architecture of Software Products*. Springer, 2012.

[9] W. Kling, F. Jouault, D. Wagelaar, M. Brambilla, and J. Cabot, "MoScript: A DSL for Querying and Manipulating Model Repositories," in *Proc. of SLE'12*. Springer, 2012, pp. 180–200.

[10] R. Salay, M. Chechik, S. Easterbrook, Z. Diskin, P. McCormick, S. Nejati, M. Sabetzadeh, and P. Viriyakattiyaporn, "An Eclipse-Based Tool Framework for Software Model Management," in *Proc. of Eclipse Workshop @ OOPSLA'07*, 2007, pp. 55–59.

[11] D. S. Kolovos, L. M. Rose, A. Garcia-Dominguez, and R. F. Paige, *The Epsilon Book*. Eclipse, 2015.

[12] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[13] J. Bézivin, F. Jouault, and D. Touzet, "An Introduction to the Atlas Model Management Architecture," Tech. Rep. 05.01, 2005.

[14] B. Vanhooff, D. Ayed, S. Van Baelen, W. Joosen, and Y. Berbers, "Uniti: A Unified Transformation Infrastructure," in *Proc. of MODELS'07*. Springer, 2007, pp. 31–45.

[15] F. Heidenreich, J. Kopcsek, and U. Aßmann, "Safe Composition of Transformations," *J. Object Technology*, vol. 7, no. 10, 2011.

[16] D. Lucrédio, R. P. d. M. Fortes, and J. Whittle, "MOOGLE: A Model Search Engine," in *Proc. of MODELS'08*. Springer, 2008, pp. 296–310.

[17] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró, "EMF-IncQuery: An Integrated Development Environment for Live Model Queries," *Science of Computer Programming*, vol. 98, pp. 80–99, 2015.

[18] M. Erwig, "Functional Programming with Graphs," *ACM SIGPLAN Notices*, vol. 32, no. 8, pp. 52–65, 1997.

[19] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Comm. of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[20] R. Lämmel, "Google's MapReduce Programming Model – Revisited," *Science of Computer Programming*, vol. 70, no. 1, pp. 1–30, 2008.