An Algebraic Framework for Merging Incomplete and Inconsistent Views

Mehrdad Sabetzadeh   Steve Easterbrook
Department of Computer Science, University of Toronto
Toronto, ON M5S 3G4, Canada.
Email: {mehrdad,sme}@cs.toronto.edu

Abstract

View merging, also called view integration, is a key problem in conceptual modeling. Large models are often constructed and accessed by manipulating individual views, but it is important to be able to consolidate a set of views to gain a unified perspective, to understand interactions between views, or to perform various types of end-to-end analysis. View merging is complicated by inconsistency of views. Once views are merged, it is useful to be able to trace the elements of the merged view back to their sources. In this paper, we propose a framework for merging incomplete and inconsistent graph-based views. We introduce a formalism, called annotated graphs, which incorporates a systematic annotation scheme capable of modeling incompleteness and inconsistency as well as providing a built-in mechanism for stakeholder traceability. We show how structure-preserving maps can capture the relationships between disparate views modeled as annotated graphs, and provide a general algorithm for merging views with arbitrary interconnections. We use the i^* modeling language [31] as an example to demonstrate how our approach can be applied to existing graph-based modeling languages.

1 Introduction

Model management is an important, but often neglected activity in requirements analysis and design. Large models are often constructed and accessed by manipulating partial views. Keeping track of the relationships between these views, and managing consistency as they evolve are major challenges [13]. Individual views may represent information from different sources, or information relevant to different development concerns. Developers analyze these views in various ways, and use the results of the analyses to improve them. Hence, individual views may evolve over time. Multiple versions of some views may be created to explore alternatives, or to respond to changing requirements. Hence, most of the time, the current set of views are likely to be incomplete and inconsistent.

In this paper, we address the problem of merging multiple views. View merging is useful in any conceptual modeling language, as a way of consolidating a set of views to gain a unified perspective, to understand interactions between views, or to perform various types of end-to-end analysis. A number of approaches for view merging have been proposed [4, 12, 23, 20]. However, all these approaches assume the set of views are consistent prior to merging them. This is fine if the views were carefully designed to work together. However, for many interesting applications, the views are not likely to be consistent *a priori*. Hence, existing approaches to view merging can only be used if considerable effort is put into detecting and repairing inconsistencies. Recent work on consistency management tools [22] helps in this respect but does not entirely address the problem because, as we will argue, it is not possible to determine whether two views are entirely consistent until all the decisions are taken about exactly how they are to be merged. The intended relationships between the views must be stated precisely.

Our approach to view merging is based on the observation that in exploratory modeling, one can never be entirely sure how concepts expressed in different views should relate to one another. Each attempt to merge a set of views can be seen as a hypothesis for how to put them together, in which choices have to be made over which concepts overlap, and how the terms used in different views relate to one another. If a particular set of choices yields an inconsistent result, it may be because we misunderstood the nature of the relationships between the views, or because there is a real disagreement between the views over either the concepts themselves, or how they are best represented. In any of these cases, it is better to perform the merge and analyze the resulting inconsistencies, rather than restrict the available merge choices.

In [26], we proposed category theory as a theoretical basis for representing structural mappings between views that contain syntactic inconsistencies. The paper proposed a systematic scheme for annotating view elements with labels denoting the amount of knowledge available about them. Relationships between views were expressed using homomorphisms that respect constraints on the annotations. View

merging was achieved by colimit computation in an appropriate category.

In this paper, we demonstrate how those ideas can be used as a framework for model management in Requirements Engineering, and to support the exploratory view merging process outlined above. We add two crucial elements: the ability to use typing information as a constraint on how views can be interconnected, and the ability to trace the elements of the merged view back to a contributing stakeholder, even when views are repeatedly elaborated. We also provide algorithms for computing the merges. To illustrate the power of the approach, we describe a novel application to the early requirements modeling language $i*$ [31].

A key assumption in our work is that the view merging problem can be studied independently of any particular conceptual modeling language. Our approach is to treat the views as structured objects, and the intended relationships between them as structural mappings. Merging views w.r.t. their interrelations can then be described using basic categorical concepts. This treatment offers both scalability to arbitrary numbers of views, and adaptability to different conceptual modeling languages.

2 Example

To motivate the paper, consider the following view merging problem. A requirements analyst, Sam, is developing a goal model for a meeting scheduler [29], based on interviews with two stakeholders, Bob and Mary. To ensure he adequately captures both contributions, Sam first models each stakeholder's view separately, using the $i*$ notation [31]. He then merges the views to study how well their goals fit together.

Figures 1(a) and 1(b) show the initial views of Mary and Bob. At first sight, there appears to be no overlap, as Mary and Bob use different terminology. However, Sam suspects there are some straight-forward relationships. Schedule meeting in Mary's view is probably the same task as Plan meeting in Bob's. Mary's Available dates be obtained may be the same goal as Bob's Responses be gathered. Sam also thinks it makes sense to treat Mary's Email requests to participants and Bob's Send request letters as alternative ways of satisfying an unstated goal, Meeting requests be sent. Bob's Consolidate results task appears to make sense as a subtask of Mary's Agreeable slot be found goal. Finally, in the light of the comparison between the views, Bob's positive contribution from Send request letters to the Efficient soft-goal looks doubtful, although the Efficient soft-goal itself seems to be important.

For a problem of this size, Sam would likely just draw his version of the merged views, with a result such as Figure 1(c), and show this to Bob and Mary for validation. This manual merge process has a number of drawbacks:

- There is no separation between hypothesizing a relationship between the original views, and generating a
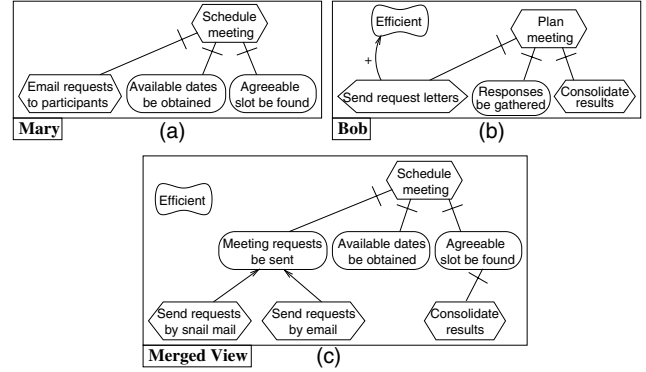

Figure 1. Motivating example

merged version based on that relationship. Hence, it is hard for Sam to test out alternative hypotheses, and it will be very hard for Bob and Mary to check Sam's assumptions individually.

- In a manual merge, Sam will naturally tend to repair inconsistencies implicitly. Hence, we lose the opportunities to analyze inconsistencies that arise with a particular choice of merge. Previous work has suggested analysis of inconsistency is a powerful means of uncovering conceptual disagreements [10].

- We have lost the stakeholders' original vocabularies. If it is important to capture the stakeholders' own vocabularies in the individual views, then it must be equally important to keep track of how those terms get adapted into the merged view.

- We have also lost the ability to trace conceptual contributions. Such traceability may become important if we repeatedly merge and evolve views over a period of time. If we later want to change the priority (or credibility!) attached to a particular stakeholder's contributions, we have no way of discovering how that stakeholder's view was incorporated into the model.

The framework we describe in this paper addresses all these problems.

3 View Merging as an Abstract Operation

The view merging framework proposed in this paper is based on a category-theoretic concept called *colimit* [1]. A *category* [1] is a collection of objects together with a collection of mappings, known as *morphisms*, between them. Each morphism expresses an admissible way to interconnect a pair of objects, showing how the structure of one object potentially maps onto the structure of another.

A hypothesis about how a group of objects are related is captured by an *interconnection diagram*. An interconnection diagram is given by a set of objects of a category and a subset of all possible morphisms between them[1]. The

---

[1] The notion of interconnection diagram in category theory is more general than this (cf. e.g. [1]), but the extra generality is unnecessary here.
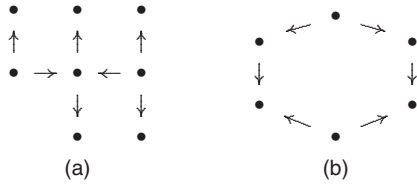
Figure 2. Examples of interconnection patterns

colimit of an interconnection diagram is a new object combining the objects in the diagram w.r.t. their relationships as declared by the morphisms in the diagram. The intuition behind colimits is that they put structures together with nothing essentially new added, and nothing left over [15]. This principle works irrespective of the exact nature of objects and morphisms.

If we wish to merge a set of views, we first need to know how they are interconnected. One of the simplest kinds of interconnection diagrams is *three-way merge*, used when we have two views A and B, along with a third view C that describes just their overlap:
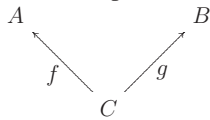
In the above interconnection diagram, two morphisms f and g specify how the common part C is represented in each of A and B. The colimit of this diagram is a new view, P, expressing the union of A and B, such that their overlap, C, is included only once.

In practice, interconnection diagrams often have more complex patterns than that of three-way merge. Figure 2 shows two examples used later in this paper: 2(a) is used for capturing the relationships between the i^* meta-model fragments in Figure 7, and 2(b) is used for capturing the relationships between the views in Figure 9.

It can be shown that each of the merge algorithms sketched in this paper corresponds to colimit computation in an appropriate category given by the respective class of objects and morphisms. Further details about the correspondence between colimits and the algorithms given herein can be found in a technical report [27] where the mathematical underpinnings of our work have been documented.

4 Interconnecting and Merging Graphs

In our framework, we assume that the underlying syntactic structure of each view can be treated as a graph. This section introduces graphs, and describes how they can be interconnected and merged. Further, it explains how graphs can be equipped with a typing mechanism. The merge algorithm for graphs is built upon that for sets; therefore, we begin with a discussion of how sets can be merged.

4.1 Merging Sets

A system of interconnected sets is given by an interconnection diagram whose objects are sets and whose mor-

---

SET-MERGE $(S_1, \ldots, S_n, f_1, \ldots, f_k)$:

▷ Let U be an initially discrete graph with node-set $S_1 \uplus \ldots \uplus S_n$;
▷ For every function f_i $(1 \leq i \leq k)$:
  ▷ For every element a in the domain of f_i:
    ▷ Add to U an undirected edge between the elements corresponding to a and $f_i(a)$;
▷ Let P be the set of the connected components of U;
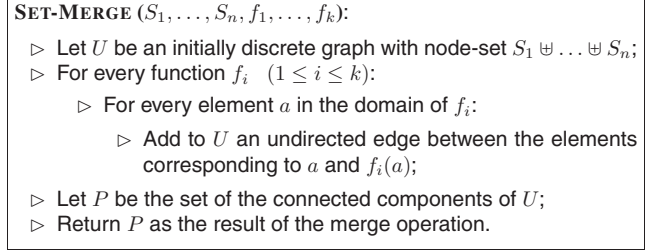▷ Return P as the result of the merge operation.

Figure 3. Algorithm for merging sets

phisms are (total) functions. Rather than treating functions as general mapping rules between arbitrary sets, we consider each function to be a mapping with a unique domain and a unique codomain. Each function can be thought of as an embedding: each element of the domain set is mapped to a corresponding element in the codomain set. For example, in a three-way merge, the morphisms would show how the set C is embedded in each of A and B.

To describe the algorithm for merging sets, we need to introduce the concept of *disjoint union*: The disjoint union of a given family of sets $S_1, S_2, \ldots, S_n$, denoted $S_1 \uplus S_2 \uplus \ldots \uplus S_n$, is (isomorphic to) the following: $S_1 \times \{1\} \cup S_2 \times \{2\} \cup \ldots \cup S_n \times \{n\}$. For conciseness, we construct the disjoint union by subscripting the elements of each given set with the name of the set and then taking the union. For example, if $S_1 = \{x, y\}$ and $S_2 = \{x, t\}$, we write $S_1 \uplus S_2$ as $\{x_{S_1}, y_{S_1}, x_{S_2}, t_{S_2}\}$ instead of $\{(x, 1), (y, 1), (x, 2), (t, 2)\}$.

To merge a system of interconnected sets, we start with the disjoint union as the largest possible merged set, and refine it by grouping together elements that get unified by the interconnections. To identify which elements should be unified, we construct a unification graph[2] U induced on the elements of the disjoint union by the interconnections; and then, combine the elements that fall in the same connected component of U. Figure 3 shows the merge algorithm for an interconnection diagram whose objects are sets $S_1, \ldots, S_n$ and whose morphisms are functions $f_1, \ldots, f_k$.

Figure 4 shows an example of three-way merge for sets: 4(a) shows the interconnection diagram; 4(b) shows the induced unification graph and its connected components; and 4(c) shows the merged set. The example shows that simply taking the union of two sets A and B might not be the right way to merge them as this may cause name-clashes (e.g. according to the interconnections, the y elements in A and B are not the same although they share the same name), or duplicates for equivalent but distinctly-named elements (e.g. according to the interconnections, w in A and t in B are the same despite having distinct names).

---

[2]A unification graph is merely a representation of a reflexive binary relation and is used to better illustrate the set merging algorithm. Unification graphs have nothing to do with directed graphs introduced later in the paper.
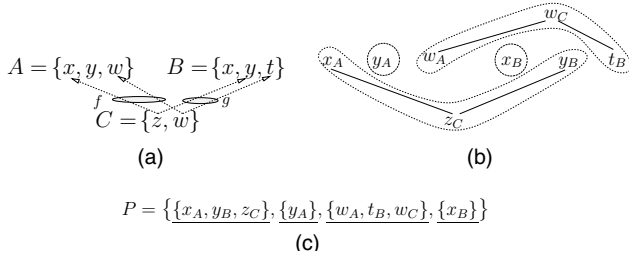
$A = \{x, y, w\}$ $B = \{x, y, t\}$
f g
$C = \{z, w\}$

(a)

(b)

$P = \{\{x_A, y_B, z_C\}, \{y_A\}, \{w_A, t_B, w_C\}, \{x_B\}\}$

(c)

Figure 4. Three-way merge example for sets

Name Mapping

To assign a name to each element of the merged set in Figure 4, we combined the names of all the elements in A, B, and C that are mapped to it. For example, "$\{x_A, y_B, z_C\}$" indicates an element that *represents* x of A, y of B, and z of C. A better way to name the elements of the merged set is assigning *naming priorities* to the input sets. For example, in three-way merge, it makes sense to give priority to the element names in the connector, C, and write the merged set in our example as $P = \{z_C, y_A, w_C, x_B\}$. Since there is no danger of name-clashes between the merged set elements in this particular example, we may choose to drop the element subscripts and write $P = \{z, y, w, x\}$.

This naming convention is of no theoretical significance; but it provides a natural solution to the name mapping problem. Generally speaking, we can assume that the choice of names in *connector objects*, i.e. objects solely used to describe the relationships between other objects, has a higher priority in determining the element names in the merged object. We will use this convention in the rest of this paper.

4.2 Graphs and Graph Merging

The notion of graph as introduced below is a specific kind of directed graph used in algebraic approaches to graph-based modeling and transformation [11], and has been successfully applied to capture various graphical formalisms including UML, Entity-Relationship Diagrams, and Petri Nets [17].

Definition 4.1 (graph) A *(directed) graph* is a tuple $G = (N, E, \mathsf{src}, \mathsf{tgt})$ where N is a set of nodes, E is a set of edges, and $\mathsf{src}, \mathsf{tgt} : E \to N$ are functions respectively giving the source and the target of each edge.

To interconnect graphs, a notion of morphism needs to be defined. A natural choice of morphism between graphs is homomorphism – a structure-preserving map describing how a graph is embedded into another:

Definition 4.2 (homomorphism) Let $G = (N, E, \mathsf{src}, \mathsf{tgt})$ and $G' = (N', E', \mathsf{src}', \mathsf{tgt}')$ be graphs. A *(graph) homomorphism* $h : G \to G'$ is a pair of functions $\langle h_{\mathbf{node}} : N \to N', h_{\mathbf{edge}} : E \to E' \rangle$ such that for all edges $e \in E$, if $h_{\mathbf{edge}}$ maps e to e' then $h_{\mathbf{node}}$ respectively maps the source and the target of e to the source and the target of e'; that is: $\mathsf{src}'(h_{\mathbf{edge}}(e)) = h_{\mathbf{node}}(\mathsf{src}(e))$
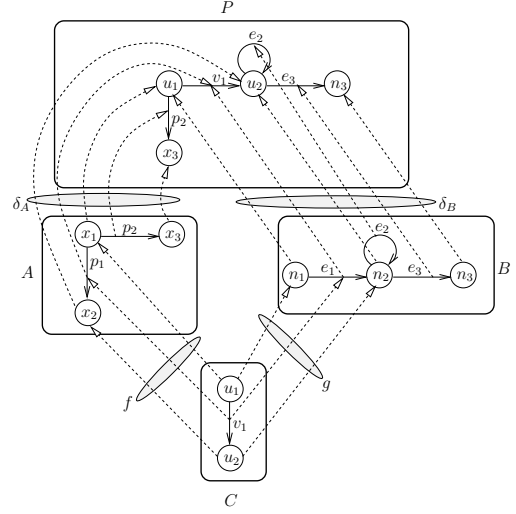

Figure 5. Three-way merge example for graphs

and $\mathsf{tgt}'(h_{\mathbf{edge}}(e)) = h_{\mathbf{node}}(\mathsf{tgt}(e))$. We call $h_{\mathbf{node}}$ the *node-map function*, and $h_{\mathbf{edge}}$ the *edge-map function* of h.

A system of interconnected graphs is given by an interconnection diagram whose objects are graphs and whose morphisms are homomorphisms. Merging is done component-wise for nodes and edges. For a graph interconnection diagram with objects $G_1, \ldots, G_n$ and morphisms $h_1, \ldots, h_k$, the merged object P is computed as follows: The node-set (resp. edge-set) of P is the result of merging the node-sets (resp. edge-sets) of $G_1, \ldots, G_n$ w.r.t. the node-map (resp. edge-map) functions of $h_1, \ldots, h_k$.

To determine the source (resp. target) of each edge e in the edge-set of the merged graph P, we pick, among $G_1, \ldots, G_n$, some graph G_i that has an edge q which is represented by e. Let s (resp. t) denote the source (resp. target) of q in G_i; and let s' (resp. t') denote the node that represents s (resp. t) in the node-set of P. We set the source (resp. target) of e in P to s' (resp. t'). Notice that an edge in the merged graph may represent edges from several input graphs. In a category-theoretic setting, it can be shown that the source and the target of each edge in the merged graph are uniquely determined irrespective of which G_i we pick.

Figure 5 shows an example of three-way merge for graphs. In the figure, each homomorphism has been visualized by a set of directed dashed lines. In addition to the homomorphisms of the interconnection diagram, i.e. f and g, we have shown the homomorphisms δ_A and δ_B specifying how A and B are represented in P. The homomorphism from C to P is implied and has not been shown.

Enforcement of Types

Graph-based modeling languages typically have typed nodes and edges. The definitions of graph and homomorphism given earlier do not support types; therefore, we need to extend them for typed graphs. We can then restrict the admissible mappings to those that preserve types.
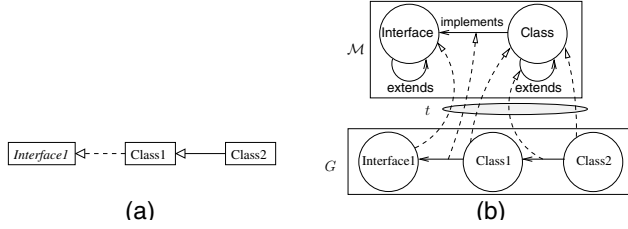
Figure 6. Example of typed graphs

In [7], a powerful typing mechanism for graphs has been proposed using the relation between the models and the meta-model for the language. Assuming that the meta-model for the language of interest is given by a graph $\mathcal{M}$, every model is described by a pair $\langle G, t : G \to \mathcal{M} \rangle$ where G is a graph and t is a homomorphism, called the *typing map*, assigning a type to every element in G. Notice that a typing map is a homomorphism, offering more structure than an arbitrary pair of functions assigning types to nodes and edges. A typed homomorphism $\underline{h} : \langle G, t \rangle \to \langle G', t' \rangle$ is simply a homomorphism $h : G \to G'$ that preserves types, i.e. $t'(h(x)) = t(x)$ for every element x in G. This typing mechanism is illustrated in Figure 6: 6(a) shows a Java class diagram in UML notation and 6(b) shows how it can be represented using a typed graph. The graph $\mathcal{M}$ in 6(b) is the extends–implements fragment of the meta-model for Java class diagrams.

The meta-model for a graph-based language can be much more complex than that of Figure 6. Figure 7 shows some fragments of the i^* meta-model extracted from the visual syntax description of i^*'s successor GRL [16]. Instead of showing the whole meta-model in one graph, we have broken it into a number of *views*, each of which represents a particular type of relationship (means-ends, decomposition, etc.). Our graph merging framework allows us to describe the meta-model without having to show it monolithically: the i^* meta-model, $\mathcal{M}_{i^*}$, is the result of merging the interconnection diagram in Figure 7. To describe the relations between the meta-model fragments, a number of connector graphs (shaded gray) have been used. Each morphism (shown by a thick solid line) is a homomorphism giving the obvious mapping. Notice that the connector graphs are *discrete* (i.e. do not have any edges) as no two meta-model fragments share common edges of the same type. The ∧- and ∨-contribution structures in i^* convey a relationship between a group of edges. To capture this, we introduced helper nodes (shown as small rectangular boxes) in the meta-model to group edges that should be related by ∧ or ∨. Structures conveying relationships between a combination of nodes and edges can be modeled similarly.

The merge operation for typed graphs is the same as that for untyped graphs. The only additional step required is assigning types to the elements of the merged graph: each element in the merged graph inherits its type from the elements it represents. In a category-theoretic setting, it can be
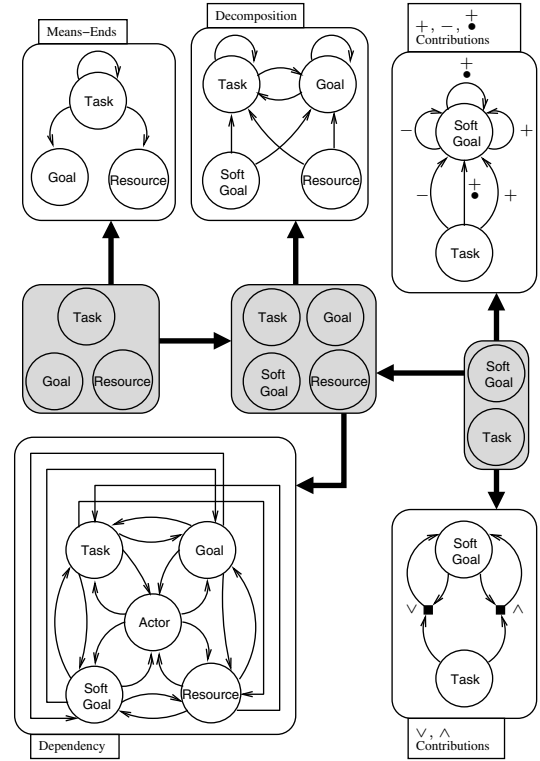

Figure 7. Some meta-model fragments of i^*

proven that every element of the merged graph is assigned a unique type in this way and that a typing map can be established from the merged graph to the meta-model.

5 Merging Requirements Views

The framework discussed in the previous section provides sufficient machinery for merging views that are free from incompleteness and inconsistency. However, as we argued earlier, for most interesting applications that involve multiple stakeholders, the views are unlikely to be either conclusive or consistent. Therefore, it is crucial to be able to tolerate incompleteness and inconsistency. In this section, we show how incompleteness and inconsistency can be modeled by an appropriate choice of *annotation* for view elements. Using the motivating example in Section 2, we demonstrate how incomplete and inconsistent views can be described, interconnected, and merged. We propose an annotation scheme which is capable of keeping track of the contributions of individual stakeholders.

5.1 Annotated Views

Consider the i^* views in Figure 1. To consolidate Mary's and Bob's views, we need to find their overlaps and, in doing so, we may need to introduce new elements (e.g. Meeting requests be sent) to properly describe the relationships between the two views. In addition, we need to capture any changes in our beliefs about the views as they evolve during the analysis. For example, we need to capture the de-
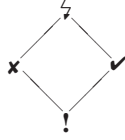
Figure 8. Variant of Belnap's knowledge order

cision that the contribution link from Sent request letters to Efficient in Bob's view is deemed ill-conceived as a result of the comparison with Mary's view.

While the classical framework of the previous section is capable of describing the correspondences between the views, it provides no means to express the stakeholders' beliefs about the *fitness* of view elements, and the possible ways in which these beliefs can *evolve*. Consequently, we cannot describe *how sure* a stakeholder is about each of the decisions he makes. We will also need to express inconsistencies that arise due to stakeholders' conflicting beliefs about the structure or the contents of views.

To model stakeholders' beliefs, we attach to each view element an annotation denoting the *degree of knowledge* available about the element. We formalize knowledge degrees using *knowledge orders*. A knowledge order is a partially ordered set [8] specifying the different levels of knowledge that can be associated to view elements, and the possible ways in which this knowledge can grow. The idea of knowledge orders was first introduced by Belnap [2], and later generalized by Ginsberg [14].

One of the simplest and arguably the most useful knowledge orders is Belnap's four-valued knowledge order [2]. The knowledge order $\mathcal{K}$ shown in Figure 8 is a variant of this: assigning **!** to an element means that the element has been *proposed* but it is not known if the element is indeed well-conceived; ✗ means that the element is known to be ill-conceived and hence *repudiated*; ✔ means that the element is known to be well-conceived and hence *affirmed*; and ⚡ means there is disagreement as to whether the element is well-conceived, i.e. the element is *disputed*.

An upward move in a knowledge order denotes a growth in the amount of knowledge, i.e. an evolution of specificity. In $\mathcal{K}$, the value **!** denotes uncertainty; ✗ and ✔ denote the conclusive amounts of knowledge; and ⚡ denotes an inconsistency, i.e. too much knowledge – we can infer something is both ill-conceived and well-conceived.

To augment graph-based views with the above-described annotation scheme, the definitions of graph and homomorphism are extended as follows. Let K be a knowledge order:

Definition 5.1 (annotated graph) A K-*annotated graph* **G** is a graph each of whose nodes and edges has been annotated with an element drawn from K.

Definition 5.2 (annotation-respecting homomorphism) Let **G** and **G**′ be K-annotated graphs. A K-*respecting homomorphism* **h** : **G** → **G**′ is a homomorphism subject

to the following condition: For every element (i.e. node or edge) x in **G**, the image of x under **h** has an annotation which is *larger than or equal to* the annotation of x.

The condition in Definition 5.2 ensures that knowledge is preserved as we traverse a morphism between annotated views. For example, if we have already decided an element in view C is *affirmed*, it cannot be embedded in another view such that it is reduced to just *proposed*, or is changed to a value not comparable to *affirmed* (i.e. *repudiated*).

For a fixed knowledge order K, the merge operation over an interconnection diagram whose objects $\mathbf{G}_1, \ldots, \mathbf{G}_n$ are K-annotated graphs and whose morphisms $\mathbf{h}_1, \ldots, \mathbf{h}_k$ are K-respecting homomorphisms, yields a merged object **P** computed as follows: First, disregard the annotations of $\mathbf{G}_1, \ldots, \mathbf{G}_n$ and merge the resulting graphs w.r.t. $\mathbf{h}_1, \ldots, \mathbf{h}_k$ to get a graph P. Then, to construct **P**, attach an annotation to every element x in P by taking the *least upper bound* [8] of the annotations of all the elements that x represents.

Intuitively, the least upper bound of a set of knowledge degrees $S \subseteq K$ is the least specific knowledge degree that refines (i.e. is more specific than) all the members of S. To ensure that the least upper bound exists for any subset of K, we assume K to be a *complete lattice* [8]. The knowledge order $\mathcal{K}$ in Figure 8 is an example of a complete lattice.

As an example, suppose the graphs in Figure 5 were annotated with $\mathcal{K}$ in such a way that the homomorphisms f and g satisfied the condition in Definition 5.2. Assuming that the nodes u_1 of C, x_1 of A, n_1 of B are respectively annotated with **!**, ✔, and ✗, the annotation for the node u_1 of P, which represents the aforementioned three nodes, is calculated by taking the least upper bound of the set $S = \{!, ✔, ✗\}$ resulting in the value ⚡.

Incorporating types into annotated graphs is independent of the annotations and is done in exactly the same manner as described in Section 4. In [27], we provide a complete version of the definitions for typed annotated graphs.

5.2 Example

We can now demonstrate how to merge the i^* views of Figure 1. We assume views are typed using the i^* meta-model $\mathcal{M}_{i^*}$ (cf. Section 4), and will use $\mathcal{K}$ (Figure 8) for annotating view elements. We therefore express relationships between views by ($\mathcal{M}_{i^*}$-typed) $\mathcal{K}$-respecting homomorphisms. Figure 9 depicts one way to express the relationships between the views in Figures 1(a) and 1(b). For convenience, we treat 'proposed' (**!**) as a default annotation for all nodes and edges, and only show annotations for the remaining values. For example, some edges in the revised versions of Bob's and Mary's views are annotated with ✗ to indicate they are repudiated.

The interconnections in Figure 9 were arrived at as follows. First, Sam creates a connector view "Connector I" to identify synonymous elements in Bob's and Mary's views.
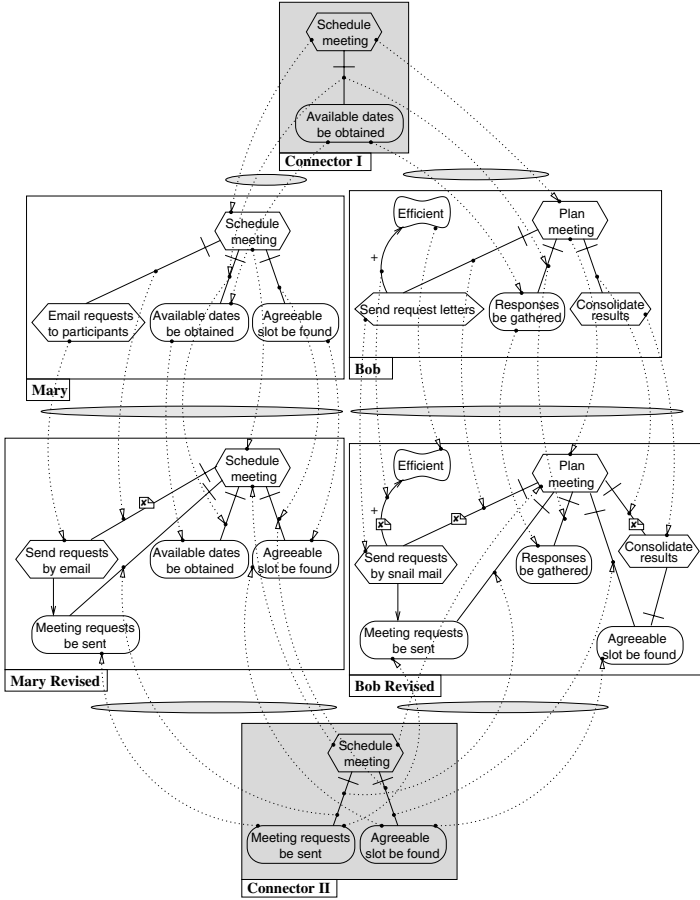
Figure 9. Interconnecting the views

To build this connector, Sam merely needs to declare which nodes in the two views are equivalent. Because i^* does not allow parallel edges of the same type between any pair of nodes, the edge interconnections are identified automatically once the node interconnections are declared. For example, when Mary's Schedule meeting and Available dates be obtained are respectively unified with Bob's Plan meeting and Responses be gathered, the decomposition links between them in the two views should also be unified.

Next, Sam elaborates each of Bob's and Mary's views to obtain "Mary Revised" and "Bob Revised". In these views, Sam has repudiated the elements he wants to replace, and proposed additional elements that he needs to complete the merge. Sam could, of course, affirm all the remaining elements of the original views, but he preferred not to do so because the models are in very early stages of elicitation. Finally, Sam identifies the common parts between the newly-added elements in the revised views, using another connector view, "Connector II".

With these interconnections, the views in Figure 9 can be automatically merged, to obtain the view shown in Figure 10. To name the elements of the merged view priority has been given to Sam's choice of names. For presentation,
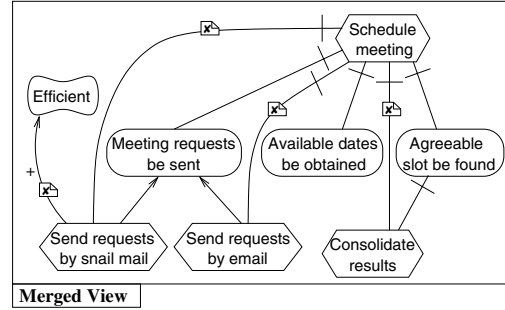

Figure 10. The merged view

we may want to *mask* the elements annotated with ✗. This would result in the view shown in Figure 1(c).

In the above scenario, we treated the original elements of Mary's and Bob's views as being at the *proposed* level, allowing Sam to freely make further decisions about any of the corresponding elements in the revised views. At any time, Mary or Bob may wish to insist upon or change their minds about any elements in their views. They can do this by elaborating their original views, affirming (or repudiating) some elements. In this case, we simply add the new elaborated views to the interconnection diagram of Figure 9, with the appropriate mappings from Mary's or Bob's original views. Such mappings are valid as long as the amount of knowledge does not decrease along morphisms. When we recompute the merge, the new annotations may result in disagreements. For example, if Mary affirms an element x in her view and Bob repudiates an element y in his, but x and y were found to be the same element by the interconnections, it would be annotated with ↯ in the merged view.

5.3 Support for Stakeholder Traceability

Direct use of $\mathcal{K}$ for annotating view elements causes two problems: firstly, every input view can include the perspective of only a single stakeholder. This is because our knowledge-ordering labels do not indicate *whose* knowledge – we have to assume all annotations within a view represent a single stakeholder. Secondly, it is not possible to distinguish the contributions of individual stakeholders in the merged view.

To provide a means for stakeholder traceability, we introduce a more elaborate annotation scheme which allows multiple stakeholders to annotate the same view: Rather than annotating view elements with single annotations, we attach an annotation-set to each element. Each annotation in the annotation-set has a qualifier denoting the stakeholder whose belief is captured by that annotation.

To ensure that the annotation-set X_e attached to a view element e evolves sanely along a morphism **h**, the following condition must hold: Every stakeholder who has an annotation in X_e must have an annotation in the annotation-set of e's image under **h**, and this annotation must be at least as specific as that in X_e. Notice that this condition does
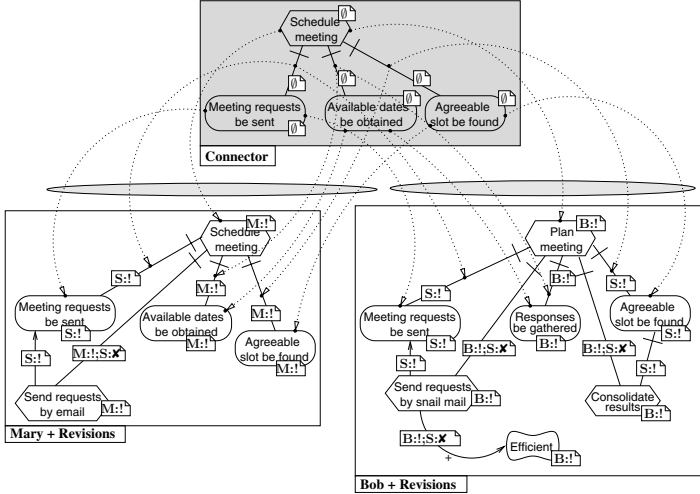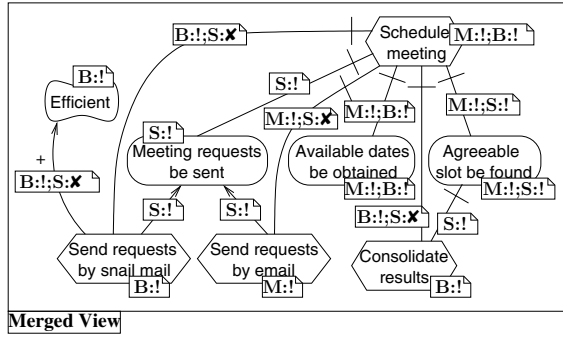
Figure 11. View interconnections

Figure 12. Merged view with detailed annotations

not prevent **h** from introducing annotations for stakeholders who do not already have an annotation in X_e – what is required is that the evolution of already-existing annotations along **h** must respect the knowledge order.

Sam may now revise the original views of Bob and Mary directly because the annotations can keep track of the contributions of individual stakeholders. The new system of interconnected views is shown in Figure 11. We use a concise notation to represent the annotation-set for each element. For example, **M:!;B:!** means that both Mary and Bob proposed the element; **B:!;S:✗** means Bob proposed the element and Sam repudiated it. Note that in the "Connector" view in Figure 11, the elements have no stakeholder annotations, indicated using ∅. If we were interested in tracking the revisions Sam makes to Bob's and Mary's vocabularies, we would need to use the same interconnection pattern as that in Figure 9, but the view elements would be annotated with annotation-sets instead of single knowledge degrees.

Merging the interconnected views in Figure 11 yields the view shown in Figure 12. The annotation-set for each view element e in the figure is computed by first unioning the stakeholders that have contributed to the elements represented by e; and then, for every stakeholder s in the union,

taking the least upper bound of s's contributions to these elements. In [27], we provide a lattice-theoretic characterization of the computation of annotation-sets using the concept of *annotated powerset lattices*.

The annotation for each element of the merged view in Figure 12 reflects the decisions made about the element by the involved stakeholders. Taking the least upper bound of the annotations in the annotation-set attached to each element of the view results in the value annotating the corresponding element in Figure 10.

Although not elaborated here due to limited space, our arguments about stakeholder traceability easily extend to other kinds of traceability including tracing elements back to the views where they were originally perceived or the domains to which the elements belong.

6 Discussion

In this section, we discuss some practical considerations concerning our proposed merge framework.

6.1 Sanity Checks

The typing mechanism discussed in Section 4 can capture many classes of typing constraints that we may have to enforce; however, it has some limitations. Most notably, it cannot capture constraints whose articulation involves multiplicities or relies on the semantics of the modeling language being used. In the class diagram example discussed in Section 4, we could not express the constraint that a Java class cannot have multiple parent classes, or that a class cannot extend its subclasses: in both cases, a typing map could be established even though the resulting class diagrams were unsound. To express the former constraint, we would have to require that the class inheritance hierarchy be a many-to-one relation; and to express the latter, we would have to require that the inheritance hierarchy be acyclic.

Finding algebraic methods for enforcing multiplicities over graphs is, to a large extent, an open problem. Some preliminary work in this direction has been reported in [18]. Dealing with constraints that require semantics-dependent reasoning has been found to be inherently non-generic [23], and hence formalism-dependent.

Due to these limitations, a number of sanity checks may be needed both on the input views, and on the merged view to ensure their soundness w.r.t. a desired set of semantic constraints. Even if the input views are sound w.r.t. such constraints, this does not imply that the merged view is sound, because the interconnections do not necessarily respect such additional constraints. If the input views are sound but the merged view is unsound, then there is a problem with how the input views have been interconnected.

Another facet to sanity checks in our framework is detecting the potential anomalies caused by annotations. For example, in Section 5, it was possible for a view to have a non-repudiated edge with a repudiated end. In such a case,

we would be left with dangling edges if we mask repudiated elements. The detection of such anomalies depends on the interpretation of the annotations being used.

6.2 Identification of Interconnections

Our focus in this paper was devising a framework for describing the relationships between incomplete and inconsistent views and merging them once the interconnections are specified. In the examples of Section 5, the interconnections were identified manually by an analyst. The natural question to ask now is to what extent we can automate the role that the analyst plays in establishing the interconnections. The answer to this question has a significant impact on how our framework scales to realistically large problems.

To our knowledge, little work has been done in Requirements Engineering on automating the identification of view interconnections, even in cases where inconsistency management has not been an issue. However, the subject has attracted considerable attention in the Database community for exploring relationships between schemata. There, the identification of interconnections is referred to as *schema matching* [24]. Unfortunately, the existing schema matching approaches are largely ad-hoc and are intertwined with the particular semantic concerns of ER diagrams. We are performing a number of case-studies on some popular graph-based formalisms including conceptual modeling languages such as i^* and (the declaration-level graphical syntax of) KAOS [30], state-machines, and UML to investigate how schema matching techniques can be generalized to graph-based structures other than ER diagrams.

6.3 Beyond Equality Relationships

To have a flexible view exploration process, we may need to express certain kinds of non-equality relationships between view elements. A notable example of such relationships is *similarity* stating that two or more elements are similar in some respects but not equivalent. Expressing non-equality relationships can be made possible by extending the formalism of interest with new modeling constructs. ER diagrams, for example, have been extended with a special notation for denoting similarities between schema elements [23]. Incorporating these relationships into merge scenarios is straight-forward. For example, in Figure 9, we could elect to introduce a similarity node instead of the Meeting request be sent goal node to state that Send request by email and Send request by snail mail are conceptually similar without providing details about the nature of the similarity.

6.4 Tool Support

We have implemented a proof-of-concept Java tool, called *iVuBlender*, for merging incomplete and inconsistent views. A brief description of the tool can be found in a demonstration paper [25] appearing in this conference proceedings.

7 Related Work

Inconsistency management has become an important topic in Requirements Engineering due to its central role in model management. A number of approaches to inconsistency management have been proposed, in general based on the success of the ViewPoints framework [13, 10, 22]. The main questions in this work center on appropriate notations for expressing consistency rules, and automated support for resolving inconsistencies. In much of this work, view merging is treated as an entirely separate problem, because of the desire to maintain viewpoints as loosely coupled distributed objects [13].

The use of multi-valued logics for merging incomplete and inconsistent views was first proposed in [9]. Our work at that time focused primarily on support for automated reasoning in the presence of inconsistency, and we developed a multi-valued model checker [5]. The central idea in this work was that merged state-machine models might contain inconsistencies, and a multi-valued model checker could be used to determine which properties are affected by the inconsistencies.

Schema merging [4, 23, 20] is an important operation in database design for combining disparate schemata, and has been identified as one of the core activities in metadata management [3, 19]. Our proposed framework extends the state-of-the-art on schema merging in several respects: firstly, the existing schema merging approaches only support the three-way merge pattern whereas our framework can handle arbitrary interconnection patterns; secondly, our framework can explicitly model inconsistencies and allows for the deferral of their resolution. This is in contrast to the above-cited work where inconsistencies are not tolerated and need to be resolved as soon as discovered. Thirdly, our framework is parameterized by a meta-model and can hence be applied to various modeling formalisms; but, schema merging is, to a great extent, tailored to ER diagrams or similar schema modeling notations.

The closest work to ours in the area of Requirements Engineering is the merging framework of [12]. In this work, views are described by graph transformation systems and colimits are used to merge them. However, the work cannot handle inconsistent views. Furthermore, the notion of view proposed therein is geared toward software specification artifacts and does not provide a suitable means for describing conceptual models.

Recently, a framework has been proposed for merging partial behavioral models [28]. There, stakeholders' models are described by partial state transition systems and are merged based on the process-algebraic refinement relations between them. The work supports incompleteness and can also detect inconsistencies; however, the merge operation fails when the models are inconsistent. Another difference between our approach and the work is that they do

not explicitly describe view correspondences and rely on bi-similarity relations to give the relationships between the states of different views. This can make it difficult for requirements analysts to guide the merge process as they cannot directly hypothesize the merge alternatives.

Annotated graphs bear similarity to fuzzy graphs [21]. The work on fuzzy graphs is focused on the analysis of isolated models using graph-theoretic techniques, whereas in our work, the focus is on describing the structural relationships between models using algebraic techniques.

8 Conclusions and Future Work

We have proposed a flexible and mathematically rigorous framework for merging incomplete and inconsistent views. Our merge framework is general and can be applied to a variety of graphical modeling languages. In this paper, we presented the core algorithms for computing merges, showed how the framework can handle typing constraints, and how our annotation scheme can be used to trace contributions in the merged view back to their sources. We have implemented the algorithms described in the paper, and used the implementation to merge views in a number of different notations.

An advantage of our framework is the explicit identification of interconnections between views prior to the merge operation rather than relying on naming conventions to give the desired unification. We believe this offers a powerful tool for exploring inconsistency during exploratory modeling, as it allows an analyst to hypothesize possible interconnections for a set of views, compute the resulting merged views, and trace between the source views and the merged views to analyze these results.

The work reported here can be continued in many directions. Automating the identification of potential interconnections between views may be an important step for applying the work to large scale conceptual modeling. Another interesting possibility is whether our framework can be used for relating the *behaviors* of models. The interconnections used in our approach are based on homomorphisms and the fact that homomorphisms have been employed in various abstraction frameworks [6] for relating behaviors of models poses many interesting questions as to what logical properties can be preserved when models are merged. Adding support for hierarchical structures is yet another issue that can be studied. We also plan to develop a more usable version of the tool to investigate how well it supports collaborative conceptual modeling, and especially stakeholder negotiation during requirements analysis.

References

[1] M. Barr and C. Wells. *Category Theory for Computing Science*. Les Publications CRM Montréal, third edition, 1999.

[2] N. Belnap. A useful four-valued logic. In *Modern Uses of Multiple-Valued Logic*, pages 5–37. Reidel, 1977.

[3] P. Bernstein. Applying model management to classical meta data problems. In *CIDR*, pages 209–220, 2003.

[4] P. Buneman et al. Theoretical aspects of schema merging. In *EDBT*, pages 152–167, 1992.

[5] M. Chechik et al. Multi-valued symbolic model-checking. *TOSEM*, 12(4):371–408, 2003.

[6] E. Clarke et al. Model checking and abstraction. *TOPLAS*, 19(2), 1994.

[7] A. Corradini et al. Graph processes. *Fundamenta Informaticae*, 26(3–4):241–265, 1996.

[8] B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, second edition, 2002.

[9] S. Easterbrook and M. Chechik. A framework for multi-valued reasoning over inconsistent viewpoints. In *ICSE*, pages 411–420, 2001.

[10] S. Easterbrook and B. Nuseibeh. Using viewpoints for inconsistency management. *Software Eng. J.*, 11:31–43, 1996.

[11] H. Ehrig and G. Taentzer. Computing by graph transformation, a survey and annotated bibliography. *BEATCS*, 59:182–226, 1996.

[12] H. Ehrig et al. A combined reference model- and view-based approach to system specification. *Intl. J. of Software Eng. and Knowledge Eng.*, 7(4):457–477, 1997.

[13] A. Finkelstein et al. Inconsistency handling in multi-perspective specifications. *TSE*, 20:569–578, 1994.

[14] M. Ginsberg. Bilattices and modal operators. In *TARK*, pages 273–287, 1990.

[15] J. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1:49–67, 1991.

[16] The GRL ontology. http://www.cs.toronto.edu/km/GRL.

[17] *Handbook of graph grammars and computing by graph transformation: volumes I–III*. World Scientific.

[18] R. Heckel and A. Zündorf. How to specify a graph transformation approach. In *ENTCS*, volume 44, 2001.

[19] S. Melnik. *Generic Model Management: Concepts and Algorithms*, volume 2967 of *LNCS*. Springer, 2004.

[20] S. Melnik et al. Rondo: a programming platform for generic model management. In *SIGMOD*, pages 193–204, 2003.

[21] J. Mordeson and P. Nair. *Fuzzy Graphs and Fuzzy Hypergraphs*. Physica-Verlag, 2000.

[22] C. Nentwich et al. Flexible consistency checking. *TOSEM*, 12:28–63, 2003.

[23] R. Pottinger and P. Bernstein. Merging models based on given correspondences. In *VLDB*, pages 862–873., 2003.

[24] E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[25] M. Sabetzadeh and S. Easterbrook. iVuBlender : A tool for merging incomplete and inconsistent views. In this proceedings.

[26] M. Sabetzadeh and S. Easterbrook. Analysis of inconsistency in graph-based viewpoints. In *ASE*, pages 12–21, 2003.

[27] M. Sabetzadeh and S. Easterbrook. An algebraic framework for merging incomplete and inconsistent views. Technical Report CSRG-496, U. of Toronto, 2004.

[28] S. Uchitel and M. Chechik. Merging partial behavioural models. In *FSE*, pages 43–52, 2004.

[29] A. van Lamsweerde et al. The meeting scheduler system – problem statement. ftp://ftp.info.ucl.ac.be/pub/publi/92.

[30] A. van Lamsweerde et al. Goal-directed elaboration of requirements for a meeting scheduler. In *RE*, pages 194–203, 1995.

[31] E. Yu. Towards modeling and reasoning support for early-phase requirements eng. In *RE*, pages 226–235, 1997.