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 incompleteness and inconsistency as well as providing a built-in mechanism for ownership traceability. We show how structure-preserving maps can capture the relationships between disparate views modeled as poset-annotated graphs, and provide a general algorithm for merging views with arbitrary interconnections. We use the i* modeling language [24] as an example to demonstrate how our approach can be applied to existing graph-based modeling languages, especially in the domain of early Requirements Engineering.

Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specifications

General Terms

Requirements Engineering, Conceptual Modeling, Viewpoints

Keywords

View Merging, Inconsistency, Traceability

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 [11]. 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 [3, 10, 19]. 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 [18] 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. Consistency checking over a set of views is only possible if the intended relationships between the views are stated precisely.

Our approach to view merging is based on the observation that in exploratory modeling, one can never be entirely sure how the 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 made poor choices, or because there is a real disagreement between the views over the nature of the concepts being modeled. In any of these cases, it is better to perform the merge and analyze the resulting inconsistencies, rather than restrict the available merge choices.

To support this exploratory process, one needs a general framework for specifying the merge choices, and computing the merges. In [21], we described such a framework for graph-based views using ideas from category theory. Briefly,
we developed 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 how the annotations can evolve. View merging was achieved by colimit computation in an appropriate category.

In this paper, we strip the framework of its categorical language, and provide algorithms for computing the merges. We present a number of important extensions, including the ability to apply typing information as a constraint on how views can be interconnected, and to trace the elements of the merged view back to the contributing stakeholders even when views are repeatedly elaborated. To illustrate the power of the approach, we describe a novel application to the early requirements modeling language $i^*$ [24].

A key assumption in our work is that the view merging problem can be studied independently of any particular conceptual modeling language. Our approach, inspired by categorical algebra [1], is to treat the views as structured objects, and the intended relationships between them as structural mappings. Merging views with their interrelations can then be described using well-known categorical concepts. This treatment offers both scalability to arbitrary numbers of views, and adaptability to different conceptual modeling languages. Applications of our approach to view merging include early requirements modeling (as illustrated in this paper), ontology integration, and schema integration.

To motivate the paper, we present a brief example of the view merging problem. Suppose a requirements analyst, Sam, is developing a requirements model for a meeting scheduler [22], 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 [24]. He then merges the views to study how well their contributions fit together.

Figures 1(a) and (b) show the initial views of Mary and Bob, expressed using $i^*$. 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. “Available dates be obtained” in Mary’s view may be the same goal as “Responses be gathered” in Bob’s. Sam also thinks it makes sense to treat “Email requests” in Mary’s view and “Send request letters” in Bob’s view as alternative ways of satisfying an unstated goal, “Meeting requests be sent”. The “Consolidate results” task in Bob’s view appears to make sense as a subtask of the “Agreeable slot be found” goal in Mary’s view. Finally, in the light of the comparison between the views, Bob’s claim of a positive contribution of “Send request letters” to the “Efficient” soft-goal looks dubious, 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 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 [9].

- 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 these problems. The remainder of the paper is structured as follows: Section 2 explains the intuition behind our proposed merge framework. Sections 3 and 4 cover the mathematical background for the paper. Section 5 introduces poset-annotated graphs, the underlying formalism for views in our framework, and describes how they can be merged. Section 6 demonstrates how poset-annotated graphs can model incompleteness and inconsistency, and provide a means for ownership traceability. Section 7 discusses some pragmatic issues concerning our framework. Section 8 presents a summary of related work. Finally, Section 9 presents conclusions and future work.

2. AN ABSTRACT MERGE FRAMEWORK

The view merging framework proposed in this paper is based on a category-theoretic concept called colimit [1]. The intuition behind colimits is that they put structures together with nothing new added, and nothing left over: “For a given species of structure, say widgets, the result of interconnecting a system of widgets to form a super-widget corresponds to taking the colimit of the diagram of widgets in which the morphisms show how they are interconnected.” [15]. This principle works irrespective of the exact nature of widgets and morphisms. In fact, each of the merge algorithms discussed in this paper corresponds to colimit computation in the respective class of widgets and morphisms.

If we wish to merge a set of views, we first need to know how they are interconnected. One of the simplest interconnection patterns is three-way merge, used when we have two
views \( A \) and \( B \), along with a third view \( C \) that describes just their common parts:

\[
\begin{array}{c}
\text{A} \\
\downarrow \\
\text{C} \\
\downarrow \\
\text{B}
\end{array}
\]

In the above interconnection diagram, two morphisms \( f \) and \( g \) specify how the common part \( C \) is represented in each of \( A \) and \( B \). Intuitively, a morphism expresses how the contents of one view are embedded in another view. The result of a three-way merge is a new view, \( P \), and three morphisms \( \delta_A : A \rightarrow P \), \( \delta_B : B \rightarrow P \), \( \delta_C : C \rightarrow P \). The view \( P \) is the union of \( A \) and \( B \) such that the common part, specified in \( C \), is included only once. The morphisms \( \delta_A \), \( \delta_B \), and \( \delta_C \) respectively show how \( A \), \( B \), and \( C \) are embedded into \( P \). In three-way merge, \( \delta_C \) can be left *implicit* because \( \delta_C = \delta_A \circ f = \delta_B \circ g \) by definition.

In practice, we may have more complex interconnection diagrams than that of three-way merge. Figure 2 shows two examples used later in this paper: 2(a) captures the relationships between the *meta-model* fragments in Figure 9 and 2(b) captures the relationships between the views in Figure 16.

![Figure 2: Examples of interconnection diagrams](image)

In general, an interconnection diagram is given by a (possibly empty) family \( w_1, \ldots, w_n \) of views; and a (possibly empty) family \( m_1: w_1 \rightarrow w_1, \ldots, m_k: w_n \rightarrow w_n \) of morphisms such that for every \( 1 \leq i \leq k \) we have: \( r_i \in \{1, 2, \ldots, n\} \) and \( t_i \in \{1, 2, \ldots, n\} \). More intuitively, an interconnection diagram is given by \( n \) views and \( k \) morphisms such that the domain (resp. the codomain) of each morphism is one of the given views. Computing the colimit of an interconnection diagram results in a view \( P \), and morphisms \( \delta_1: w_1 \rightarrow P, \ldots, \delta_n: w_n \rightarrow P \) showing how each \( w_1, \ldots, w_n \) is embedded into \( P \). We call \( P \) the *merged view* and call \( \delta_1, \ldots, \delta_n \) the *embedding maps*.

In the remainder of this paper, we will not directly use category theory. Instead, we take an algorithmic approach to explaining the merge operations and skip their category-theoretic underpinnings. [21] provides a detailed exposition of the category-theoretic details.

### 3. MERGING SETS AND GRAPHS

In this section, we explain the algorithms for merging sets and graphs. These algorithms will be referred to in later sections where we discuss merging poset-annotated sets and poset-annotated graphs.

#### 3.1 Merging Sets

A system of interconnected sets is given by a diagram in which the widgets are sets and the morphisms are total functions. Strictly speaking, a morphism uniquely identifies its domain and its codomain; in other words, we cannot treat functions as general mapping rules for arbitrary sets. In the examples given in this paper, we will represent functions by their mapping relations but we always make sure that the domain and the codomain of each function are clear from the context.

The algorithm for merging sets hinges on two basic operations: disjoint union, and the smallest equivalence relation induced by a given pair of parallel functions. We first review these operations and then give the merge algorithm for sets.

**Disjoint Union (\( \biguplus \))**

The disjoint union of a family of sets \( S_1, S_2, \ldots, S_n \), denoted \( S_1 \uplus S_2 \uplus \cdots \uplus S_n \), is (isomorphic to) the following: \( S_1 \times \{1\} \cup S_2 \times \{2\} \cup \cdots \cup S_n \times \{n\} \). For convenience, we construct the disjoint union by subscribing the elements of each 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 \( \{(x, 1), (y, 1), (x, 2), (t, 2)\} \).

In Figure 16, \( f \) and \( g \) is the union of \( A \) and \( B \) such that the common part, \( C \), is included only once. The morphisms \( \delta_A \), \( \delta_B \), and \( \delta_C \) respectively show how \( A \), \( B \), and \( C \) are embedded into \( P \). In three-way merge, \( \delta_C \) can be left *implicit* because \( \delta_C = \delta_A \circ f = \delta_B \circ g \) by definition.

In practice, we may have more complex interconnection diagrams than that of three-way merge. Figure 2 shows two examples used later in this paper: 2(a) captures the relationships between the *meta-model* fragments in Figure 9 and 2(b) captures the relationships between the views in Figure 16.

---

**Smallest Equivalence Relation (unifier)**

Given a pair of parallel functions \( f: X \rightarrow Y \) and \( g: X \rightarrow Y \), *unifier* computes the smallest equivalence relation induced by them. This is done as follows: consider an undirected graph \( G \) whose node-set is \( Y \) and whose (undirected) edge-set is given by the following relation: \( R = \{(f(a), g(a)) \mid a \in X\} \).

That is, for every \( a \in X \), there is an undirected edge between \( f(a) \) and \( g(a) \) in \( G \). Applying *unifier* to \( f \) and \( g \) yields a set \( U \) and a function \( q: Y \rightarrow U \) such that \( f(a) \) and \( g(a) \) belong to the same equivalence class of \( q \) if and only if \( a \) belongs to the same equivalence class.

---

**The Merge Algorithm**

The idea behind the merge algorithm is simple: given an interconnected system of sets, we initially start with the disjoint union of all the sets as the largest possible merged set and iteratively refine it by grouping the elements that get unified by the interconnections. This is continued until all the interconnections are accounted for.

Let \( S_1, S_2, \ldots, S_k \) be sets; and let \( f_1: S_1 \rightarrow T_1, \ldots, f_k: S_k \rightarrow T_k \) be functions where \( r_i \in \{1, 2, \ldots, n\} \) and \( t_i \in \{1, 2, \ldots, n\} \) for all \( 1 \leq i \leq k \). The result of merging \( S_1, \ldots, S_k \) w.r.t. \( f_1, \ldots, f_k \) is the set \( P \) along with the functions \( \delta_1: S_1 \rightarrow P, \ldots, \delta_n: S_n \rightarrow P \) as computed by the algorithm in Figure 3.

---

**Figure 4** shows an example of three-way merge for sets when \( A = \{x, y, w\} \), \( B = \{x, y, t\} \), and \( C = \{z, w\} \) with \( f: C \rightarrow A \) and \( g: C \rightarrow B \) given as follows: \( f = \{(x, x) \rightarrow x, (y, w) \rightarrow w\} \) and \( g = \{(z, z) \rightarrow z, (y, t) \rightarrow t\} \). In Figure 5, we illustrate execution of the merge algorithm on our example: 5(a) shows the configuration just before entering the unification loop; and 5(b)–(c) respectively show the configurations after the first and the second iteration of this loop. We have used superscripts for \( P, q, \delta_1, \delta_2 \), and \( \delta_3 \) to differentiate between the values of the variables at each stage. The algorithm starts with
Merge-Sets:
\[ P = S_1 \sqcup \ldots \sqcup S_n; \] /* initialize the merged set to the disjoint union */
for \( i = 1 \) to \( n \) do
\[ \delta_i = \alpha_i; \] /* initialize the embedding map at \( S_i \) to the injection at \( S_i \) */
end
for \( j = 1 \) to \( k \) do /* the unification loop */
\[ P, q = \text{unifier}(\delta_j \circ f_j, \delta_j); \] /* \( f_j \) has \( S_j \) as domain and \( S_j \) as codomain */
end
for \( i = 1 \) to \( n \) do
\[ \delta_i = q \circ \delta_i; \] /* adjust the embedding map at \( S_i \) */
end

Figure 3: Algorithm for merging sets
\[
P = \{ (x_A, y_B, z_C), (y_A), \{ w_A, t_B, w_C \}, \{ x_B \} \}
\]
\[ A = \{ (y, y), (y, y) \} \]
\[ B = \{ (x, x) \} \]
\[ C = \{ (z, z) \} \]

Figure 4: Three-way merge example for sets
\[ P^0 = A \sqcup B \sqcup C. \]
In the first iteration, \( P^1 \) and \( q^1 \) are computed by a call to \( \text{unifier}(\delta_B^0 \circ f, \delta_C^0); \) and in the second iteration \( P^2 \) and \( q^2 \) are computed by a call to \( \text{unifier}(\delta_B^0 \circ g, \delta_C^0). \)
The embedding maps are adjusted accordingly in each iteration of the unification loop. Note that nesting caused by successive unifications can be ignored. Therefore, we wrote the merged set as \( \{ (x_A, y_B, z_C), (y_A), \{ w_A, t_B, w_C \}, \{ x_B \} \} \) instead of \( \{ (x_A, y_B, z_C), (y_A), \{ w_A, w_C, t_B \}, \{ x_B \} \} \) — the two sets are isomorphic.

This example clearly shows that simply taking the union of two sets \( A \) and \( B \) is not the right way to merge them because this may cause name-clashes (e.g. according to the interconnections, the 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 although they have distinct names).

Name Mapping
For assigning a name to each element of the merged set in Figure 4, we took the union of the names of all those elements in \( A, B, \) and \( C \) that are mapped to it. For example, the name \( \{ (x_A, y_B, z_C) \} \) indicates that the element is the image of \( x \) in \( A, y \) in \( B \) and \( z \) in \( C \). A more concise way to name the elements of the merged set is giving priority to the element names in \( C \) because \( C \) plays the role of a connector.

Merge-Sets:
\[ P = S_1 \sqcup \ldots \sqcup S_n; \]
foreach \( i \) in \( n \) do
\[ \delta_i = \alpha_i; \]
foreach \( j \) in \( k \) do /* the unification loop */
\[ P, q = \text{unifier}(\delta_j \circ f_j, \delta_j); \]
foreach \( i \) in \( n \) do
\[ \delta_i = q \circ \delta_i; \]
end
end

Figure 5: Execution of the merge algorithm

Compute-src-tgt:
\[
\text{for } i = 1 \to n \text{ do }
\]
\[
\text{for every edge } e \in G, \text{ do }
\]
\[ \text{src} = (\gamma(e) = \beta(\text{src}(e)); \]
\[ \text{tgt} = (\gamma(e) = \beta(\text{tgt}(e)); \]
end
end

Figure 6: Computation of \( \text{src} \) and \( \text{tgt} \)

between \( A \) and \( B \). Then, we may write the merged set in our example as \( \{ z_C, y_A, w_C, x_B \} \). This assumption is of no theoretical significance; but, it can be viewed as an ad-hoc solution to the name mapping problem. Generally speaking, we can assume that the choice of names in \( \text{connector} \) objects, i.e. objects solely used to describe the relationships between other objects, has a higher priority in determining the names of elements in the merged object. In the rest of this paper, with the exception of Figure 12, we will use this assumption for naming the elements of merged objects. The assumption should, of course, be used with care because it may give rise to contention between connectors or even name-clashes.

3.2 Merging Graphs
In this subsection, we introduce graphs and explain how they can be merged.

Definition 3.1 A graph is a tuple \( G = (N, E, \text{src}, \text{tgt}) \) where \( N \) is a set of nodes, \( E \) is a set of edges, and \( \text{src} : E \to N \) are functions respectively giving the source and the target of each edge.

Definition 3.2 Let \( G = (N, E, \text{src}, \text{tgt}) \) and \( G' = (N', E', \text{src}', \text{tgt}') \) be graphs. A graph homomorphism \( h : G \to G' \) is a pair of functions \( h = (h_{\text{node}} : N \to N', h_{\text{edge}} : E \to E') \) such that for all \( e \in E \), if \( e' \) is the image of \( e \) under \( h_{\text{edge}} \) then the source and the target of \( e' \) are respectively the images of the source and the target of \( e \) under \( h_{\text{node}} \):
\[
h_{\text{node}} \circ \text{src} = \text{src}', h_{\text{node}} \circ \text{tgt} = \text{tgt}'.
\]
We call \( h_{\text{node}} \) the node-map component and call \( h_{\text{edge}} \) the edge-map component of \( h \).

Merging graphs is done component-wise for nodes and edges: let \( G_1 = (N_1, E_1, \text{src}_1, \text{tgt}_1), \ldots, G_n = (N_n, E_n, \text{src}_n, \text{tgt}_n) \) be graphs; and let \( h_i : G_i \to G_k \) be graph homomorphisms with \( r_i \in \{1, 2, \ldots, n\} \) and \( t_i \in \{1, 2, \ldots, n\} \) for all \( 1 \leq i \leq k \).

Let \( N_P \) and \( E_P \) along with \( h_{\text{node}} : N \to N', h_{\text{edge}} : E \to E' \) be the result of merging \( N_1, \ldots, N_n \) w.r.t. the node-map components of \( h_1, \ldots, h_k \); let \( E_{\ell} \) along with \( h_{\text{node}} : N \to N', h_{\text{edge}} : E \to E' \) be the result of merging \( E_1, \ldots, E_n \) w.r.t. the edge-map components of \( h_1, \ldots, h_k \); and let the functions \( \text{src}_P, \text{tgt}_P : E_P \to N_P \) be given by the algorithm in Figure 6.

The result of merging \( G_1, \ldots, G_n \) w.r.t. \( h_1, \ldots, h_k \) is the graph \( P = (N_P, E_P, \text{src}_P, \text{tgt}_P) \), along with the graph homomorphisms \( h_i = (\beta_i, \gamma_i) \) mapping each \( G_i \) to \( P \). Outside a categorical setting, it is not trivial to prove that \( \text{src}_P \) and \( \text{tgt}_P \), as computed by the algorithm make \( P \) a graph and make \( \beta_1, \ldots, \beta_k \) graph homomorphisms. Figure 7 shows an example of three-way merge for graphs.

Enforcement of Types
Graph-based modeling languages typically have typed nodes and edges. The definitions of graph and graph homomorphism given earlier do not support types; therefore, we need to augment them for typed graphs. We can then restrict the admissible homomorphisms to those that preserve types. In
Figure 7: Example of merging graphs

Figure 8: Example of typed graphs

Figure 9: Some meta-model fragments of $i^*$

are discrete since no two meta-model fragments share common edges of the same type.

The $\wedge$- and $\lor$-contribution structures in $i^*$ convey a relationship between a group of edges. To capture this, we introduced helper nodes (shown as solid boxes) in the meta-model to group edges that should be related by $\wedge$ or $\lor$. Structures conveying relationships between a combination of nodes and edges can be modeled similarly.

Merging Typed Graphs

The typing mechanism just described provides a way to assign types to graph elements, and makes the interconnections more structured by requiring them to preserve types. The merge operation for typed graphs is exactly 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 has the same type as that of the elements mapped to it by the embedding maps. In a category-theoretic setting, it can be 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.

4. ANNOTATING SETS WITH POSETS

In this section, we review poset-annotated sets and explain how they can be merged. Poset-annotated sets serve as the basic building blocks for poset-annotated graphs to be introduced in the next section. In addition, we discuss the notion of poset-annotated powerset which will be exploited in Section 6 for devising a traceability mechanism.

4.1 Lattice Theory

We assume the reader is familiar with basic concepts of partially-ordered sets (posets) and lattices. Below, we briefly review some definitions and lemmas to establish notation. An excellent introduction to partial orders and lattices is [7].

Definition 4.1 Let $Q$ be a poset; and let $A \subseteq Q$. An element $q \in Q$ is an upper bound of $A$ if $\forall a \in A : a \leq q$. If
Figure 10: The lattice \( Q \)

\( q \) is an upper bound of \( A \) and \( q \leq Q x \) for all upper bounds \( x \) of \( A \), then \( q \) is called the supremum of \( A \). Dually, an element \( q \in Q \) is a lower bound of \( A \) if \( \forall a \in A : q \leq a \). If \( q \) is a lower bound of \( A \) and \( x \leq q \) for all lower bounds \( x \) of \( A \), then \( q \) is called the infimum of \( A \). We write \( \bigsqcup Q A \) (resp. \( \bigsqcap Q A \)) to denote the supremum (resp. infimum) of \( A \subseteq Q \), when it exists.

**Definition 4.2** Let \( Q \) be a poset. If both \( \bigsqcup Q \{ a, b \} \) and \( \bigsqcap Q \{ a, b \} \) exist for any \( a, b \in Q \), then \( Q \) is called a lattice. If both \( \bigsqcup Q A \) and \( \bigsqcap Q A \) exist for any \( A \subseteq Q \), then \( Q \) is called a complete lattice.

**Lemma 4.3** Every complete lattice has a bottom (\( \bot \)) and a top (\( \top \)) element. Every finite lattice is complete.

4.2 Poset-Annotated Sets

A poset-annotated set is a set whose elements are annotated with values drawn from a partial order. The definitions given in this subsection are borrowed from [14] where fuzzy set theory has been formulated in a category-theoretic setting. We prefer to use the term “poset-annotated” rather than “fuzzy” to signify the fact that the posets we deal with in our work differ in nature from the linearly ordered real interval \([0,1]\) commonly used in fuzzy set theory.

**Definition 4.4** Let \( Q \) be a poset. A \( Q \)-annotated set is a pair \((S, \sigma)\) consisting of a set \( S \) and a function \( \sigma : S \to Q \). We call \( S \) the carrier set of \((S, \sigma)\) and call \( Q \) the annotation universe of \( \sigma \).

**Definition 4.5** Let \( Q \) be a poset; and let \((S, \sigma)\) and \((T, \tau)\) be \( Q \)-annotated sets. A \( Q \)-respecting map \( f : (S, \sigma) \to (T, \tau) \) is a function \( f : S \to T \) such that \( \sigma \leq Q \tau \circ f \), i.e. for every \( s \in (S, \sigma) \), the \( Q \)-value annotating \( s \) is no larger than the \( Q \)-value annotating \( f(s) \) in \((T, \tau)\). The function \( f : S \to T \) is called the carrier function of \( f \).

As an example, suppose the annotation universe is the lattice \( Q \) in Figure 10. Then, \( \{(1, 1), (2, 2)\}, \{2 \to a, 3 \to b\} \) and \( \{(1, 1), (1, 2), (1, 3) \to d, (2, 2) \to b, (3, 3) \to c\} \) are \( Q \)-annotated sets and the function \( f : \{1 \to t_1, 2 \to t_2\} \) is an \( Q \)-respecting map because \( \sigma(1) \leq Q \tau(f(1)) \) and \( \sigma(2) \leq Q \tau(f(2)) \).

4.3 Merging Poset-Annotated Sets

The merge operation for an arbitrary system of interconnected \( Q \)-annotated sets requires that the supremum exist for every \( A \subseteq Q \) (cf. [21]). To ensure that \( Q \) has this property, we assume \( Q \) is a complete lattice.

Let \((S_1, \sigma_1), \ldots, (S_n, \sigma_n)\) be \( Q \)-annotated sets; and let \( f_1 : (S_1, \sigma_1) \to (S_1, \sigma_1), \ldots, f_n : (S_n, \sigma_n) \to (S_n, \sigma_n)\) be \( Q \)-respecting maps with \( r_i \in \{1, 2, \ldots, n\} \) and \( t_i \in \{1, 2, \ldots, n\} \) for all \( 1 \leq i \leq k \).

Let the set \( P \) along with the functions \( \delta_1, \ldots, \delta_k \) be the result of merging \( S_1, \ldots, S_n \) w.r.t. the carrier functions of \( f_1, \ldots, f_n \); and let the function \( \rho : P \to Q \) be given by the algorithm in Figure 11. The result of merging \((S_1, \sigma_1), \ldots, (S_n, \sigma_n)\) w.r.t. \( f_1, \ldots, f_n \) is the \( Q \)-annotated set \((P, \rho)\) along with \( \delta_1, \ldots, \delta_n \), when viewed as \( Q \)-respecting maps.

Figure 12 shows an example of three-way merge for \( Q \)-annotated sets where \( Q \) is the lattice shown in Figure 10.

**Figure 11:** Computation of \( \rho \)

\[
\begin{align*}
\text{compute}\rho: & \\
\text{for every } p \in P \text{ do} & \rho(p) = \bot_Q; (* \text{ initialize } \rho(p) \text{ to } \bot_Q \*) \\
\text{od} & \\
\text{for } i = 1 \text{ to } n \text{ do} & \\
\text{for every } s \in S_i \text{ do} & \rho(\delta_i(s)) = \bigsqcup Q \{\rho(\delta_i(s)), \sigma_i(s)\}; \\
\text{od} & \\
\text{od} & \\
\end{align*}
\]

**Figure 12:** Example of merging poset-annotated sets

The carrier function of each \( Q \)-respecting map \( f, g, \delta_A, \delta_B, \) and \( \delta_C \) is the same as the corresponding function in Figure 5. The function \( \rho \) has been computed by the algorithm in Figure 11. For example, the annotation for the element \( \{x_A, y_B, z_C\} \) is \( \bigsqcup Q \{\delta_A(x), \sigma_B(y), \sigma_C(z)\} = d \).

To make Figure 12 concise, we have depicted every annotated set \((S, \sigma)\) as a set of tuples \( \{(s, \sigma(s)) \mid s \in S\} \). The same convention will be used in Figure 13.

4.4 Poset-Annotated Powersets

In Section 6, we will develop an annotation scheme which allows decorating each view element with the decisions of multiple stakeholders. Assuming that the decisions are drawn from a poset \( Q \), the annotation for each view element can be represented by a \( Q \)-annotated set \((S, \sigma)\) where \( S \) denotes the set of involved stakeholders and \( \sigma : S \to Q \) associates a \( Q \)-value to every \( s \in S \). To describe the annotation universe for this annotation scheme, we need to introduce the notion of poset-annotated powerset:

**Definition 4.6** Let \( Q \) be a poset; and let \((Z, \mu)\) be a \( Q \)-annotated set. The \( Q \)-annotated powerset induced by \((Z, \mu)\) is the set of all \( Q \)-annotated sets \((S, \sigma)\) such that \( S \subseteq Z \) and for every \( s \in S \) we have: \( \sigma(s) \leq \mu(s) \).

**Lemma 4.7** [14] When \( Q \) is a complete lattice, so is the powerset induced by any \( Q \)-annotated set.

**Proof (sketch)** [14] For an index set \( I \), the supremum of an \( I \)-indexed family \( \{(S_i, \sigma_i)\}_{i \in I}\) of powerset elements is a \( Q \)-annotated set \((X, \theta)\) where \( X = \bigcup_{i \in I} S_i \) and \( \theta : X \to Q \) is a function such that for every element \( x \in X \) we have: \( \theta(x) = \bigsqcup Q \{\sigma_i(x) \mid x \in S_i; i \in I\} \). The infimum is computed dually.

As an example, suppose \( Q \) is the two-point lattice \( \{0, 1\} \) with \( 0 < 1 \). Then, the \( Q \)-annotated powerset induced by \( \{(x, y); \{x \mapsto 0, y \mapsto 1\}\} \) is the lattice shown in Figure 13.

5. ANNOTATING GRAPHS WITH POSETS

This section introduces poset-annotated graphs and gives a merge algorithm for them. Poset-annotated graphs augment typed graphs by adding poset annotations to them. The key observation making it possible to use poset-annotated graphs for modeling incompleteness and inconsistency is the capability of posets to capture a notion of “knowledge degree” and possible ways in which knowledge can evolve. This concept will be explained and exemplified in Section 6.
5.1 Poset-Annotated Graphs

Let $I$ and $J$ be fixed posets; and let $\mathcal{M}$ be a fixed graph.

**Definition 5.1** An $\mathcal{M}$-typed $(I,J)$-annotated graph $G$ is a tuple $((N, \sigma), (E, \tau), \text{src}, \text{tgt}, t)$ where

- $(N, \sigma : N \rightarrow I)$ is an $I$-annotated set of nodes;
- $(E, \tau : E \rightarrow J)$ is a $J$-annotated set of edges;
- $\text{src}, \text{tgt} : E \rightarrow N$ are functions respectively giving the source and the target of each edge;
- $t$, called the **typing map**, is a graph homomorphism from $G = ((N, E, \text{src}, \text{tgt})$ to $\mathcal{M}$.

We call $G = ((N, E, \text{src}, \text{tgt})$ the carrier graph of $G$.

**Definition 5.2** Let $G$ and $G'$ be $(I,J)$-annotated graphs. An $\mathcal{M}$-typed $(I,J)$-**respecting homomorphism** $h : G \rightarrow G'$ is a pair $h = (\text{hnode}, \text{hedge})$ where $\text{hnode}$ is an $I$-respecting map from the node-set of $G$ to the node-set of $G'$ and $\text{hedge}$ is a $J$-respecting map from the edge-set of $G$ to the edge-set of $G'$ such that the pair $h = (\text{hnode}, \text{hedge})$ given by the carrier functions of $\text{hnode}$ and $\text{hedge}$ has the following properties:

- $h$ is a graph homomorphism from the carrier graph of $G$ to that of $G'$;
- $h$ preserves types, that is: $h \circ t' = t$ where $t$ and $t'$ are respectively the typing maps of $G$ and $G'$.

We call $\text{hnode}$ the **node-map component** and call $\text{hedge}$ the **edge-map component** of $h$. The pair $h = (\text{hnode}, \text{hedge})$ is called the **carrier homomorphism** of $h$.

5.2 Merging Poset-Annotated Graphs

Merging poset-annotated graphs is similar to merging graphs in that it is done component-wise for nodes and edges. The difference is that the node-sets and edge-sets are poset-annotated sets rather than plain sets. To ensure soundness of the merge operation for the node-sets and the edge-sets of poset-annotated graphs, we require that the annotations be complete lattices (cf. [21]). In the rest of this section, we assume that poset-annotated graphs are untyped. Extending the arguments to the typed case is identical to that for plain graphs (cf. Subsection 3.2).

Let $I$ and $J$ be fixed complete lattices. Let $G_t = \text{(}(N_t, \sigma_t), (E_t, \tau_t), \text{src}_t, \text{tgt}_t)\text{)}$, ..., $G_n = \text{(}(N_n, \sigma_n), (E_n, \tau_n), \text{src}_n, \text{tgt}_n)\text{)}$ be $(I,J)$-annotated graphs; and let $h_1 : G_1 \rightarrow G_t$, ..., $h_n : G_n \rightarrow G_t$ be $(I,J)$-respecting homomorphisms with $r_i \in \{1, 2, \ldots, n\}$ and $t_i \in \{1, 2, \ldots, k\}$ for all $1 \leq i \leq k$.

Let the $I$-annotated set $(N_P, \sigma_P)$ along with the $I$-respecting maps $\beta_1, \ldots, \beta_n$ be the result of merging $(N_1, \sigma_1), \ldots, (N_n, \sigma_n)$ w.r.t. the node-map components of $h_1, \ldots, h_n$; let the $J$-annotated set $(E_P, \tau_P)$ along with the $J$-respecting maps $\gamma_1, \ldots, \gamma_k$ be the result of merging $(E_1, \tau_1), \ldots, (E_n, \tau_n)$ w.r.t. the edge-map components of $h_1, \ldots, h_n$; and let the functions $\text{src}_P, \text{tgt}_P : E_P \rightarrow N_P$ be given by the algorithm in Figure 6 by letting each $\beta_i$ (resp. $\gamma_i$) be the carrier function of $\beta_i$ (resp. $\gamma_i$) for $1 \leq i \leq n$.

The result of merging $G_1, \ldots, G_n$ w.r.t. $h_1, \ldots, h_n$ is the $(I,J)$-annotated graph $P = ((N_P, \sigma_P), (E_P, \tau_P), \text{src}_P, \text{tgt}_P)$ along with the $(I,J)$-respecting homomorphisms $\delta_1 = (\beta_1, \gamma_1), \ldots, \delta_n = (\beta_n, \gamma_n)$ mapping every $G_i$ to $P$. Proving that $P$ is an $(I,J)$-annotated graph and $\delta_1, \ldots, \delta_n$ are $(I,J)$-respecting homomorphisms is non-trivial outside a category-theoretic context (cf. [21]).

Figure 14 shows an example of three-way merge for untyped $(Q, Q)$-annotated graphs where $Q$ is the lattice shown in Figure 10. The carrier homomorphism of each $f, g, \delta_A$, and $\delta_B$ is the same as the corresponding graph homomorphism in Figure 7. The annotation associated with each element is attached to it in an annotation box.

6. MERGING REQUIREMENTS VIEWS

In this section, we show how incompleteness and inconsistency can be modeled by an appropriate choice of annotation universe. Using the motivating problem in the introduction, we demonstrate how partial and inconsistent views can be described, interconnected, and merged. Furthermore, we show how the results in Section 4.4 can be used as an annotation mechanism that makes it possible to trace the elements of the merged view back to the contributing stakeholder(s).

For modeling incompleteness and inconsistency, we use a “knowledge-ordering” lattice as the annotation universe. A knowledge-ordering lattice allows us to specify the amount of knowledge we have about each view element. The idea of knowledge ordering was first described by Belnap [2], and later generalized by Ginsberg [12].

One of the simplest and arguably the most useful knowledge-ordering lattices is Belnap’s four-valued lattice [2]. The lattice $K$ shown in Figure 15 is a variant of this: assigning $I$ to an element means that the element has been proposed but it is not known if the element is indeed well-conceived; $\not I$ means that the element is known to be ill-conceived and hence repudiated; $\check{I}$ means that the element is known to be well-conceived and hence affirmed; and $\check{I}$ means there is disagreement as to whether the element is well-conceived, i.e. the element is disputed.

An upward move in a knowledge-ordering lattice denotes an increase in the amount of knowledge. In $K$, the value $I$ denotes uncertainty; $\not I$ and $\check{I}$ denote the desired (i.e. decisive) amounts of knowledge; and $\check{I}$ denotes an inconsistency, i.e. too much knowledge – we can infer something is both ill-conceived and well-conceived.

As an example, we show how to merge the $i^*$ views of Fig-
Figure 15: The knowledge-ordering lattice $\mathcal{K}$

Figure 16: Interconnecting the views

Figure 17: The merged view

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 16 can be automatically merged, to obtain the view shown in Figure 17. For presentation, we may want to mask the elements annotated with $\otimes$. 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, affixing (or repudiating) some elements. In this case, we simply add the new elaborated views to the interconnection diagram of Figure 16, 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 $\uparrow$ in the merged view.

Direct use of $\mathcal{K}$ for annotating the view elements causes two problems: first, 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. Second, it is not possible to distinguish the contributions of individual stakeholders in the merged view. To address these problems, we introduce a more elaborate annotation scheme that allows multiple stakeholders to annotate the same view.

Consider the $\mathcal{K}$-annotated powerset $\mathcal{P}$ induced by $\{(\text{Sam, Bob, Mary}), (\text{Sam, Bob, Mary})\}$ (cf. Definition 4.6). Since $\mathcal{K}$ is a complete lattice, by Lemma 4.7, $\mathcal{P}$ is also a complete lattice. This allows us to describe the views by $\mathcal{M}_{\ast\ast}$-typed $(\mathcal{P}, \mathcal{P})$-annotated graphs and use $\mathcal{M}_{\ast\ast}$-typed $(\mathcal{P}, \mathcal{P})$-respecting homomorphisms to interconnect them. Sam may now revise the original views of Bob and Mary directly because the annotations keep track of the decision each stakeholder makes about an element. The new system of interconnected views is shown in Figure 18. We use a concise notation to represent the $\mathcal{P}$-value annotating each element. For example, $\mathbf{B}:!;\mathbf{S}:\mathbf{X}$ means...
Figure 18: Interconnecting powerset-annotated views

Figure 19: The powerset-annotated merged view

7.1 Sanity Checks

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 [17]. Dealing with constraints that require semantics-dependent reasoning has been found to be inherently non-generic [19], 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 is detecting the potential anomalies caused by the annotations. For example, in Section 6, 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 elements of the annotation universes being used.

7.2 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 example in Section 6, 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 merging database schemata [20]. There, the identification of interconnections is referred to as schema matching. The existing schema matching techniques are mostly suited to the three-way merge pattern, but we believe they can be generalized to arbitrary interconnection patterns without much change. However, schema matching is largely ad-hoc and 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 [23], state-machines, and UML to investigate how schema matching techniques can be tailored to graph-based structures other than ER diagrams.

7.3 Tool Support

We have implemented a proof-of-concept Java tool for
merging views. The tool is essentially a library for computing colimits in poset-annotated graph categories. We have used the tool for merging $\mathcal{I}^*$ models, ER diagrams, and state-machines. The tool can also be used for developing algebraic graph transformation systems for poset-annotated graphs, but we have not explored this application yet.

8. 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 [11, 9, 18]. 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 [11]. The closest work to ours in this area is the graph-transformation-based framework of [13]. There, inconsistencies are defined explicitly in terms of the relations that should hold, and graph rewrite rules are used to detect and repair them.

The use of multi-valued logics for merging inconsistent views was first proposed in [8]. 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 [4]. 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.

In [10], a colimit-based approach has been given for merging consistent views. In that approach, viewpoints are described by graph transformation systems and colimits are employed to integrate them. However, this work cannot handle inconsistent views.

Merging is crucial to the Database community for schema integration [3, 19]. This is the subject of an active research area. Our approach generalizes the syntactic aspects of the merge operation proposed in the cited references.

9. 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 area is studying whether our framework can be used for relating the behaviors of models. The interconnections used in our approach are based on graph homomorphisms and the fact that graph homomorphisms have been employed in various abstraction frameworks [5] 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 area that can be studied. We also plan to develop a more useable version of the tool to investigate how it supports collaborative conceptual modeling, and especially stakeholder negotiation during requirements analysis.

Acknowledgments. We thank John Mylopoulos for suggesting the example and Linda Liu for help with the $\mathcal{I}^*$ notation. We thank the members of the Formal Methods, Database, and the EarlyRE groups at the University of Toronto for their insightful comments on this work. Financial support was provided by NSERC, MITACS, and Bell University Labs.

10. REFERENCES