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

September 2004

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 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 *poset-annotated graphs*, which incorporates a systematic annotation scheme capable of modeling 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 [26] 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.

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.

In [22], 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 source view, 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**** [26].

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 w.r.t. 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 [24], 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 [26]. 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 straightforward 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

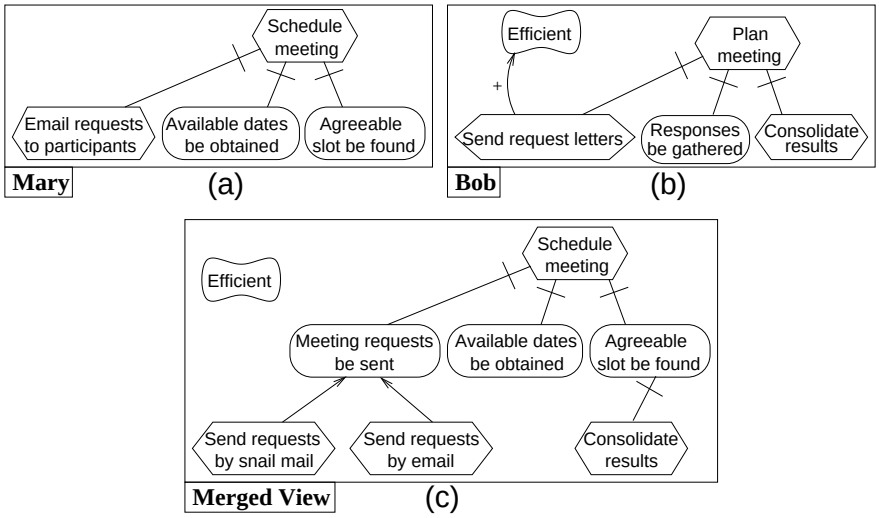


Figure 1: Motivating example

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 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.

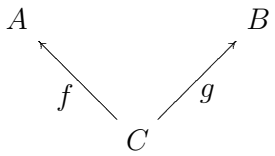
The framework we describe in this paper addresses these problems. The remainder of the paper is structured as follows: Section 2 explains the intuition behind our proposed merge framework. Sections 3 and 4 cover the mathematical background for the paper. Section 5 introduces poset-annotated graphs, the 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." [14]. 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:



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 $\delta_A : A \to P$, $\delta_B : B \to P$, $\delta_C : C \to P$. The view P is the union of A and B such that the common part, specified in C, is included only once. The morphisms δ_A, δ_B, and δ_C respectively show how A, B, and C are embedded into P. In three-way merge, δ_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 $\boldsymbol{i}^*$ meta-model fragments in Figure 10 and 2(b) captures the relationships between the views in Figure 18.

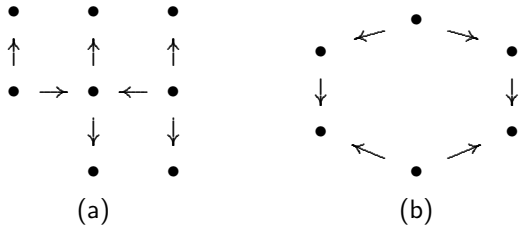


Figure 2: Examples of interconnection diagrams

In general, an interconnection diagram[1] is given by a (possibly empty) family $w_1, \ldots, w_n$ of views; and a (possibly empty) family $m_1 : w_{r_1} \to w_{t_1}, \ldots, m_k : w_{r_k} \to w_{t_k}$ 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 $\delta_1 : w_1 \to P, \ldots, \delta_n : w_n \to P$ 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. [22] provides a detailed exposition of the category-theoretic details. Merging sets (Subsection 3.1) is based on the constructive proof of finite cocompleteness for an arbitrary category [1, 21]; merging graphs and typed graphs (Subsection 3.2) is based on colimit construction in comma categories [21]. Merging poset-annotated sets (Subsection 4.3) is based on colimit construction in indexed categories [23] by noticing that for a poset Q,

[1] The notion of interconnection diagram in category theory is slightly more general than what is given here (cf. e.g. [1]), but the extra generality is unnecessary in our work.

the category of Q-annotated sets is isomorphic to the category of discrete diagrams in Q. Merging poset-annotated graphs (Subsection 5.2) is based on colimit construction in comma categories and noting the colimit preservation property given by Lemma 3.6 in [22].

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 $S_1 \uplus S_2 \uplus \ldots \uplus S_n$, is (isomorphic to) the following: $S_1 \times \{1\} \cup S_2 \times \{2\} \cup \ldots \cup S_n \times \{n\}$. For convenience, we construct the disjoint union by subscripting 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 $\{(x,1),(y,1),(x,2),(t,2)\}$. For each S_i, $1 \leq i \leq n$, there is a function $\alpha_i : S_i \to S_1 \uplus S_2 \uplus \ldots \uplus S_n$, called the *injection at* S_i, mapping every element of S_i to its image in the disjoint union. In the above example, the injection at S_1 is the function $\alpha_1 = \{x \mapsto x_{S_1}, y \mapsto y_{S_1}\}$.

Smallest Equivalence Relation (*unifier*)

Given a pair of parallel functions $f : X \to Y$ and $g : X \to Y$, ***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: $R = \big\{\big(f(a), g(a)\big) \mid a \in X\big\}$. 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 $q : Y \to U$ where U is the (canonical) set of G's connected components and $q : Y \to U$ is the function taking every $y \in Y$ to the connected component of G to which y belongs. Strictly speaking, ***unifier*** need only yield q because every function uniquely identifies its domain and codomain; however, we found it more intuitive to include U as an output of ***unifier***.

As an example, let $X = \{a, b, c\}$ and $Y = \{u, v, w, t\}$; and let functions $f : X \to Y$ and $g : X \to Y$ be given as follows: $f = \{a \mapsto u, b \mapsto v, c \mapsto w\}$ and $g = \{a \mapsto u, b \mapsto w, c \mapsto w\}$. Then, ***unifier***$(f, g)$ yields $U = \big\{\{u\}, \{v, w\}, \{t\}\big\}$ and $q = \big\{u \mapsto \{u\}, v \mapsto \{v, w\}, w \mapsto \{v, w\}, t \mapsto \{t\}\big\}$. The induced graph G is depicted in Figure 3.

Figure 3: Computation of connected components

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 $S_1, \ldots, S_n$ be sets; and let $f_1 : S_{r_1} \to S_{t_1}, \ldots, f_k : S_{r_k} \to S_{t_k}$ 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 $S_1, \ldots, S_n$ w.r.t. $f_1, \ldots, f_k$ is the set P along with the functions $\delta_1 : S_1 \to P, \ldots, \delta_n : S_n \to P$ as computed by the algorithm in Figure 4.

MERGE-SETS:

$P = S_1 \uplus \ldots \uplus S_n$; /* 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 */
od

for $j = 1$ **to** k **do** /* the unification loop */
  $P, q = \textbf{\textit{unifier}}(\delta_{t_j} \circ f_j, \delta_{r_j})$; /* f_j has S_{r_j} as domain and S_{t_j} as codomain */
  for $i = 1$ **to** n **do**
    $\delta_i = q \circ \delta_i$; /* adjust the embedding map at S_i */
  od
od

Figure 4: Algorithm for merging sets

An important property of the merge algorithm in Figure 4 is that the merged set and the embedding functions it generates are insensitive (up to isomorphism) to the orderings on $S_1, \ldots, S_n$ and $f_1, \ldots, f_k$. This results from the *universality* [1] of colimits.

Figure 5 shows an example of three-way merge for sets when $A = \{x, y, w\}$, $B = \{x, y, t\}$, and $C = \{z, w\}$ with $f : C \to A$ and $g : C \to B$ given as follows: $f = \{z \mapsto x, w \mapsto w\}$ and $g = \{z \mapsto y, w \mapsto t\}$.

In Figure 6, we illustrate execution of the merge algorithm on our example: 6(a) shows the configuration just before entering the unification loop; and 6(b)–(c) respectively show the configurations after the first and the second iteration of this loop. We have used superscripts for P, q, δ_A, δ_B, and δ_C to differentiate between the values of the variables at each stage. The algorithm starts with $P^0 = A \uplus B \uplus C$. In the first iteration, P^1 and q^1 are computed by a call to $\textbf{\textit{unifier}}(\delta_A^0 \circ f, \delta_C^0)$; and in the second iteration P^2 and q^2 are computed by a call to $\textbf{\textit{unifier}}(\delta_B^1 \circ g, \delta_C^1)$. The embedding maps are adjusted accordingly in each iteration of the unification loop. Note that nesting caused by successive unifications can

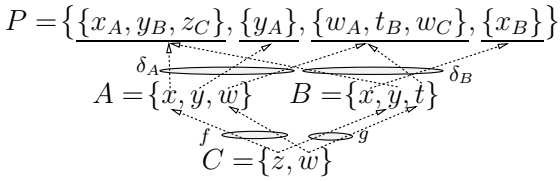


Figure 5: Three-way merge example for sets

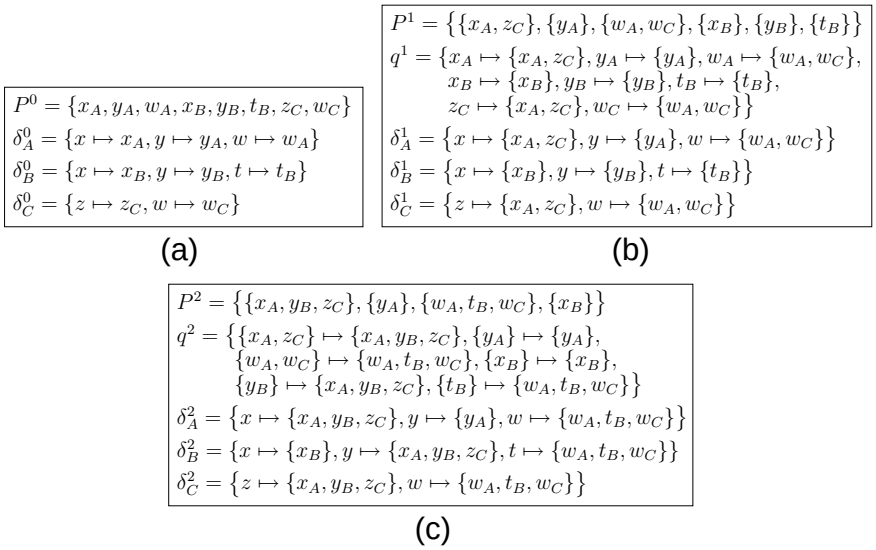


Figure 6: Execution of the merge algorithm

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 $\{\{\{x_A, z_C\}, \{y_B\}\}, \{\{y_A\}\}, \{\{w_A, w_C\}, \{t_B\}\}, \{\{x_B\}\}\}$ – 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 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 although they have distinct names).

Name Mapping

For assigning a name to each element of the merged set in Figure 5, 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 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 *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 14, we will use this assumption for naming the elements of merged objects.

```
COMPUTE-src-tgt:
  for i = 1 to n do
      for every edge e in G_i do
          src_P(γ_i(e)) = β_i(src_i(e));
          tgt_P(γ_i(e)) = β_i(tgt_i(e));
      od
  od
```

Figure 7: Computation of src_P and tgt_P

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 $G = (N, E, \mathsf{src}, \mathsf{tgt})$ 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.

Definition 3.2 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 $h = \langle h_{\mathbf{node}} : N \to N', h_{\mathbf{edge}} : E \to E' \rangle$ such that for all $e \in E$, if e' is the image of e under $h_{\mathbf{edge}}$ then the source and the target of e' are respectively the images of the source and the target of e under $h_{\mathbf{node}}$; that is: $h_{\mathbf{node}} \circ \mathsf{src} = \mathsf{src}' \circ h_{\mathbf{edge}}$ and $h_{\mathbf{node}} \circ \mathsf{tgt} = \mathsf{tgt}' \circ h_{\mathbf{edge}}$.

We call $h_{\mathbf{node}}$ the *node-map component* and call $h_{\mathbf{edge}}$ the *edge-map component* of h.

Merging graphs is done component-wise for nodes and edges: let $G_1 = (N_1, E_1, \mathsf{src}_1, \mathsf{tgt}_1), \ldots, G_n = (N_n, E_n, \mathsf{src}_n, \mathsf{tgt}_n)$ be graphs; and let $h_1 : G_{r_1} \to G_{t_1}, \ldots, h_k : G_{r_k} \to G_{t_k}$ 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 along with $\beta_1 : N_1 \to N_P, \ldots, \beta_n : N_n \to N_P$ 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_P along with $\gamma_1 : E_1 \to E_P, \ldots, \gamma_n : E_n \to E_P$ 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 $\mathsf{src}_P, \mathsf{tgt}_P : E_P \to N_P$ be given by the algorithm in Figure 7.

The result of merging $G_1, \ldots, G_n$ w.r.t. $h_1, \ldots, h_k$ is the graph $P = (N_P, E_P, \mathsf{src}_P, \mathsf{tgt}_P)$ along with the graph homomorphisms $\delta_1 = \langle \beta_1, \gamma_1 \rangle, \ldots, \delta_n = \langle \beta_n, \gamma_n \rangle$ mapping every G_i to P. Outside a categorical setting, it is not trivial to prove that src_P and tgt_P as computed by the algorithm make P a graph and make $\delta_1, \ldots, \delta_n$ graph homomorphisms. Figure 8 shows an example of three-way merge for graphs. In the figure, we have visualized every graph homomorphism by drawing directed dashed lines from the elements of the source graph to their images in the target graph.

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 [6], a powerful typing mechanism for graphs has been proposed using the relation between the models in a graph-based language 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 graph homomorphism, called the *typing map*, assigning a type to every element in G. Notice that a typing map is a graph homomorphism and hence more structured than an arbitrary pair of functions one assigning types to

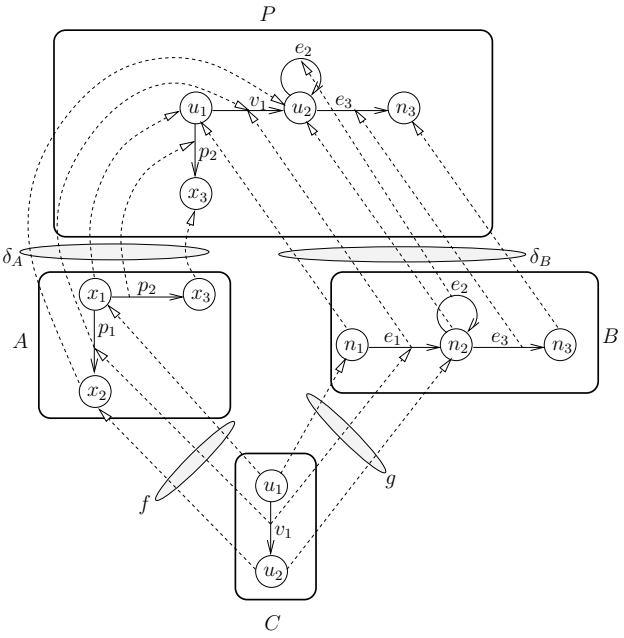


Figure 8: Example of merging graphs

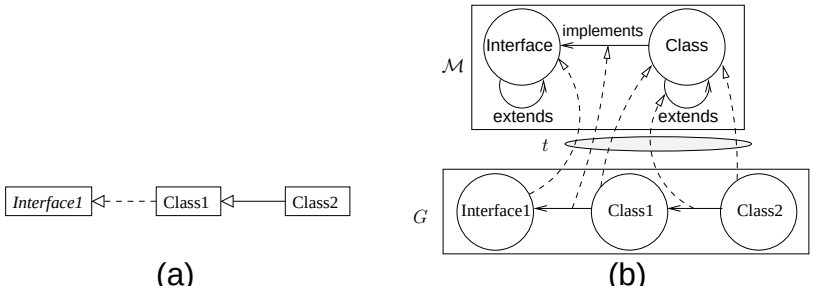


Figure 9: Example of typed graphs

nodes and the other assigning types to edges. A typed graph homomorphism $\underline{h} : \langle G, t\rangle \rightarrow \langle G', t'\rangle$ is simply a graph homomorphism $h : G \rightarrow G'$ that preserves types, that is: $t' \circ h = t$.

This typing mechanism is illustrated in Figure 9: 9(a) shows a Java class diagram in UML notation and 9(b) shows how it can be represented using a typed graph. The graph $\mathcal{M}$ in 9(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 for the class diagram example in Figure 9. Figure 10 shows some fragments of the ***i**** meta-model extracted from the visual syntax description of ***i****'s successor GRL [15]. Rather than showing the whole meta-model in one graph, we have broken it into a number of views each of which describes a particular type of relationship (means-ends, decomposition, etc.) Our graph merging framework described above allows us to describe the meta-model without having to show it monolithically: the ***i**** meta-model, denoted $\mathcal{M}_{\boldsymbol{i}*}$, is the result of merging the interconnection diagram in Figure 10. To describe the relations between the meta-model fragments, a number of connector graphs (shaded grey) have been used. Each morphism (shown by a thick solid line) is a graph homomorphism giving the obvious mapping. Notice that the connector graphs are *discrete* since no two

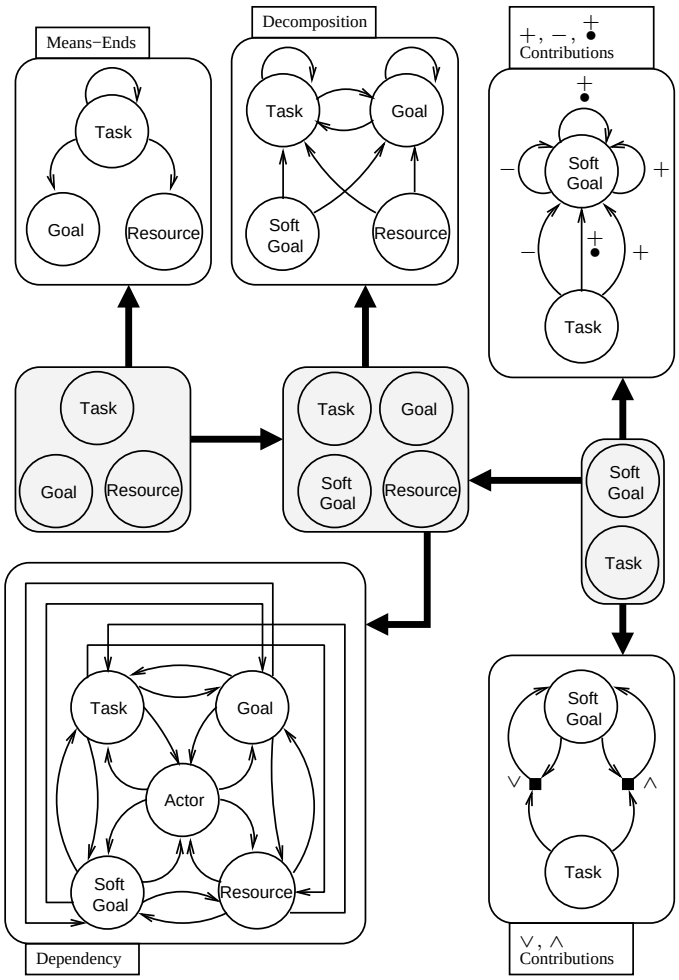


Figure 10: Some meta-model fragments of ***i****

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 solid 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. Figure 11(a) shows how we normally draw an ∨-contribution structure in an ***i**** model and Figure 11(b) shows the adaptation of the structure to typed graphs. Relationships between nodes and edges are 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.

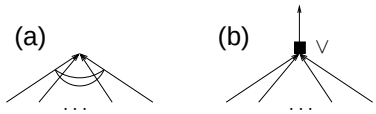


Figure 11: Adaptation of ∨-contribution

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 $\forall a \in A : a \leq_Q q$. If 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 $\forall a \in A : q \leq_Q a$. If q is a lower bound of A and $x \leq_Q q$ 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 [13] 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, σ) consisting of a set S and a function $\sigma : S \to Q$. We call S the *carrier set* of (S, σ) and call Q the *annotation universe* of σ.

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

As an example, suppose the annotation universe is the lattice $\mathcal{Q}$ in Figure 12. Then, $\big(\{s_1, s_2\}, \{s_1 \mapsto a, s_2 \mapsto b\}\big)$ and $\big(\{t_1, t_2, t_3\}, \{t_1 \mapsto d, t_2 \mapsto b, t_3 \mapsto c\}\big)$ are $\mathcal{Q}$-annotated sets and the function $f : \{s_1 \mapsto t_1, s_2 \mapsto t_2\}$ is an $\mathcal{Q}$-respecting map because $\sigma(s_1) \leq_{\mathcal{Q}} \tau(f(s_1))$ and $\sigma(s_2) \leq_{\mathcal{Q}} \tau(f(s_2))$.

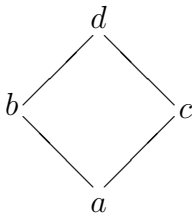


Figure 12: The lattice $\mathcal{Q}$

```
COMPUTE-ρ:
  for every p ∈ P do
      ρ(p) = ⊥_Q;     /* initialize ρ(p) to ⊥_Q */
  od

  for i = 1 to n do
      for every s ∈ S_i do
          ρ(δ_i(s)) = ⊔_Q{ρ(δ_i(s)), σ_i(s)};
      od
  od
```

Figure 13: Computation of ρ

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. [22]). 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 $\mathbf{f}_1 : (S_{r_1}, \sigma_{r_1}) \to (S_{t_1}, \sigma_{t_1}), \ldots, \mathbf{f}_k : (S_{r_k}, \sigma_{r_k}) \to (S_{t_k}, \sigma_{t_k})$ 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 $\delta_1, \ldots, \delta_n$ be the result of merging $S_1, \ldots, S_n$ w.r.t. the carrier functions of $\mathbf{f}_1, \ldots, \mathbf{f}_k$; and let the function $\rho : P \to Q$ be given by the algorithm in Figure 13. The result of merging $(S_1, \sigma_1), \ldots, (S_n, \sigma_n)$ w.r.t. $\mathbf{f}_1, \ldots, \mathbf{f}_k$ is the Q-annotated set (P, ρ) along with $\delta_1, \ldots, \delta_n$ when viewed as Q-respecting maps.

Figure 14 shows an example of three-way merge for $\mathcal{Q}$-annotated sets where $\mathcal{Q}$ is the lattice shown in Figure 12. The carrier function of each $\mathcal{Q}$-respecting map $\mathbf{f}$, $\mathbf{g}$, $\boldsymbol{\delta_A}$, $\boldsymbol{\delta_B}$, and $\boldsymbol{\delta_C}$ is the same as the corresponding function in Figure 6.

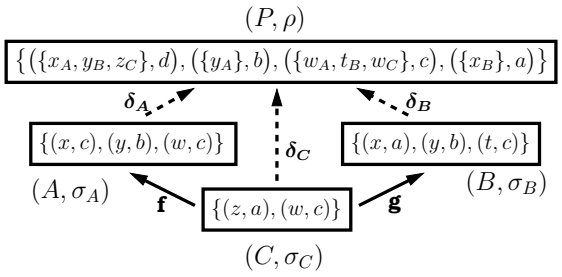


Figure 14: Example of merging poset-annotated sets

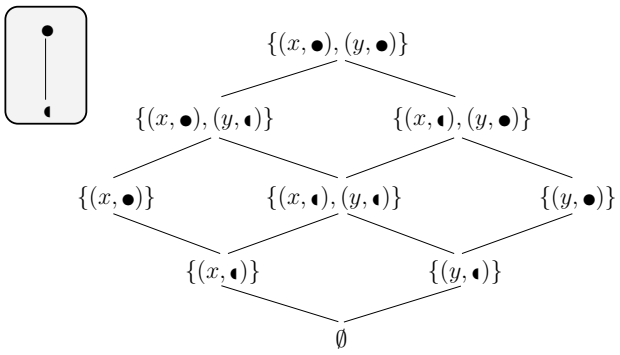


Figure 15: Example of powerset lattice

The function ρ has been computed by the algorithm in Figure 13. For example, the annotation for the element "$\{x_A, y_B, z_C\}$" is $\bigsqcup_{\mathcal{Q}}\{\bot_{\mathcal{Q}}, \sigma_A(x), \sigma_B(y), \sigma_C(z)\} = d$.

To make Figure 14 concise, we have depicted every annotated set (S, σ) as a set of tuples $\big\{\big(s, \sigma(s)\big) \mid s \in S\big\}$. The same convention will be used in Figure 15.

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

Lemma 4.7 *[13] When Q is a complete lattice, so is the powerset induced by any Q-annotated set.*

Proof (*sketch*) [13] For an index set I, the supremum of an I-indexed family $\big\langle (S_i, \sigma_i)\big\rangle_{i\in I}$ of powerset elements is a Q-annotated set (X, θ) 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 $\{◖, ●\}$ with $◖ < ●$. Then, the Q-annotated powerset induced by $\big(\{x, y\}, \{x \mapsto ●, y \mapsto ●\}\big)$ is the lattice shown in Figure 15.

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* $\mathbf{G}$ is a tuple $\big((N, \sigma), (E, \tau), \mathsf{src}, \mathsf{tgt}, t\big)$ where

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

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

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

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

We call $\mathbf{h}_{\mathbf{node}}$ the *node-map component* and call $\mathbf{h}_{\mathbf{edge}}$ the *edge-map component* of $\mathbf{h}$. The pair $h = \langle h_{\mathbf{node}}, h_{\mathbf{edge}} \rangle$ is called the *carrier homomorphism* of $\mathbf{h}$.

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 annotation universes be complete lattices (cf. [22]). 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 $\mathbf{G}_1 = \big((N_1, \sigma_1), (E_1, \tau_1), \mathsf{src}_1, \mathsf{tgt}_1\big), \ldots, \mathbf{G}_n = \big((N_n, \sigma_n), (E_n, \tau_n), \mathsf{src}_n, \mathsf{tgt}_n\big)$ be (I, J)-annotated graphs; and let $\mathbf{h}_1 : \mathbf{G}_{r_1} \to \mathbf{G}_{t_1}, \ldots, \mathbf{h}_k : \mathbf{G}_{r_k} \to \mathbf{G}_{t_k}$ be (I, J)-respecting 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 the I-annotated set (N_P, σ_P) along with the I-respecting maps $\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_n$ be the result of merging $(N_1, \sigma_1), \ldots, (N_n, \sigma_n)$ w.r.t. the node-map components of $\mathbf{h}_1, \ldots, \mathbf{h}_k$; let the J-annotated set (E_P, τ_P) along with the J-respecting maps $\boldsymbol{\gamma}_1, \ldots, \boldsymbol{\gamma}_n$ be the result of merging $(E_1, \tau_1), \ldots, (E_n, \tau_n)$ w.r.t. the edge-map components of $\mathbf{h}_1, \ldots, \mathbf{h}_k$; and let the functions $\mathsf{src}_P, \mathsf{tgt}_P : E_P \to N_P$ be given by the algorithm in Figure 7 by letting each β_i (resp. γ_i) be the carrier function of $\boldsymbol{\beta_i}$ (resp. $\boldsymbol{\gamma_i}$) for $1 \leq i \leq n$.

Merging $\mathbf{G}_1, \ldots, \mathbf{G}_n$ w.r.t. $\mathbf{h}_1, \ldots, \mathbf{h}_k$ yields $\mathbf{P} = \big((N_P, \sigma_P), (E_P, \tau_P), \mathsf{src}_P, \mathsf{tgt}_P\big)$ along with the (I, J)-respecting homomorphisms $\boldsymbol{\delta}_1 = \langle \boldsymbol{\beta}_1, \boldsymbol{\gamma}_1 \rangle, \ldots, \boldsymbol{\delta}_n = \langle \boldsymbol{\beta}_n, \boldsymbol{\gamma}_n \rangle$ mapping every $\mathbf{G}_i$ to $\mathbf{P}$. Proving that $\mathbf{P}$ is an (I, J)-annotated graph and $\boldsymbol{\delta}_1, \ldots, \boldsymbol{\delta}_n$ are (I, J)-respecting homomorphisms is non-trivial outside a category-theoretic context (cf. [22]).

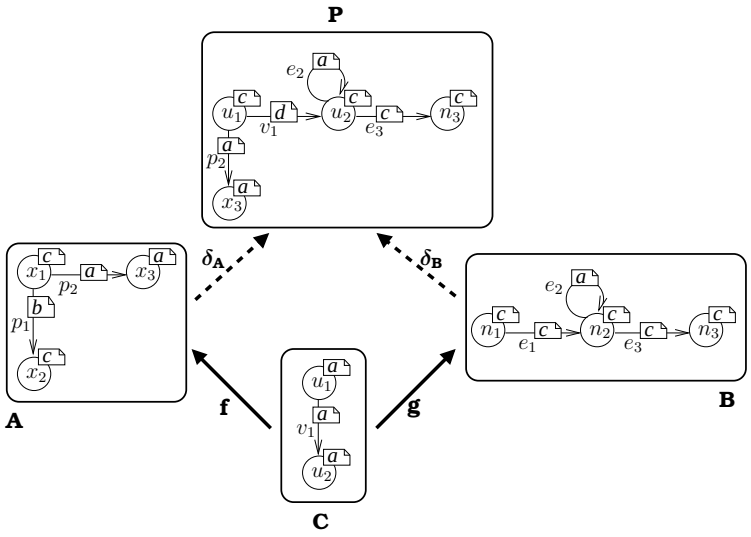


Figure 16: Example of merging poset-annotated graphs

Figure 16 shows an example of three-way merge for untyped $(\mathcal{Q}, \mathcal{Q})$-annotated graphs where $\mathcal{Q}$ is the lattice shown in Figure 12. The carrier homomorphism of each $\mathbf{f}$, $\mathbf{g}$, $\boldsymbol{\delta}_\mathbf{A}$, and $\boldsymbol{\delta}_\mathbf{B}$ is the same as the corresponding graph homomorphism in Figure 8. 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 $\mathcal{K}$ shown in Figure 17 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 ϟ 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 $\mathcal{K}$, the value **!** denotes uncertainty; ✘ and ✔ denote the desired (i.e. decisive) amounts of knowledge; and ϟ denotes an inconsistency, i.e. too much knowledge – we can infer something is both ill-conceived and well-conceived.

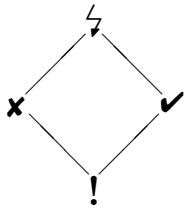


Figure 17: The knowledge-ordering lattice $\mathcal{K}$

As an example, we show how to merge the i^* views of Figure 1. We assume these views are typed using the i^* meta-model, $\mathcal{M}_{i^*}$, and we will annotate nodes and edges of the views using $\mathcal{K}$. We therefore express relationships between views using $\mathcal{M}_{i^*}$-typed $(\mathcal{K}, \mathcal{K})$-respecting homomorphisms. Figure 18 depicts one way to express the relationships between Mary's and Bob's 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 and Mary's views are annotated with ✘ to indicate they are repudiated.

The interconnections in Figure 18 were arrived at as follows. First, Sam creates a connector view "Connector I" to identify synonymous elements in Mary's and Bob's 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 "Schedule meeting" and "Available dates be obtained" in Mary's view are respectively unified with "Plan meeting" and "Responses be gathered" in Bob's view, it can be inferred that 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 18 can be automatically merged, to obtain the view shown in Figure 19. For presentation, 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 18, 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 ϟ 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. This effectively makes it infeasible to perform any type of analysis over the merged view that needs to distinguish between

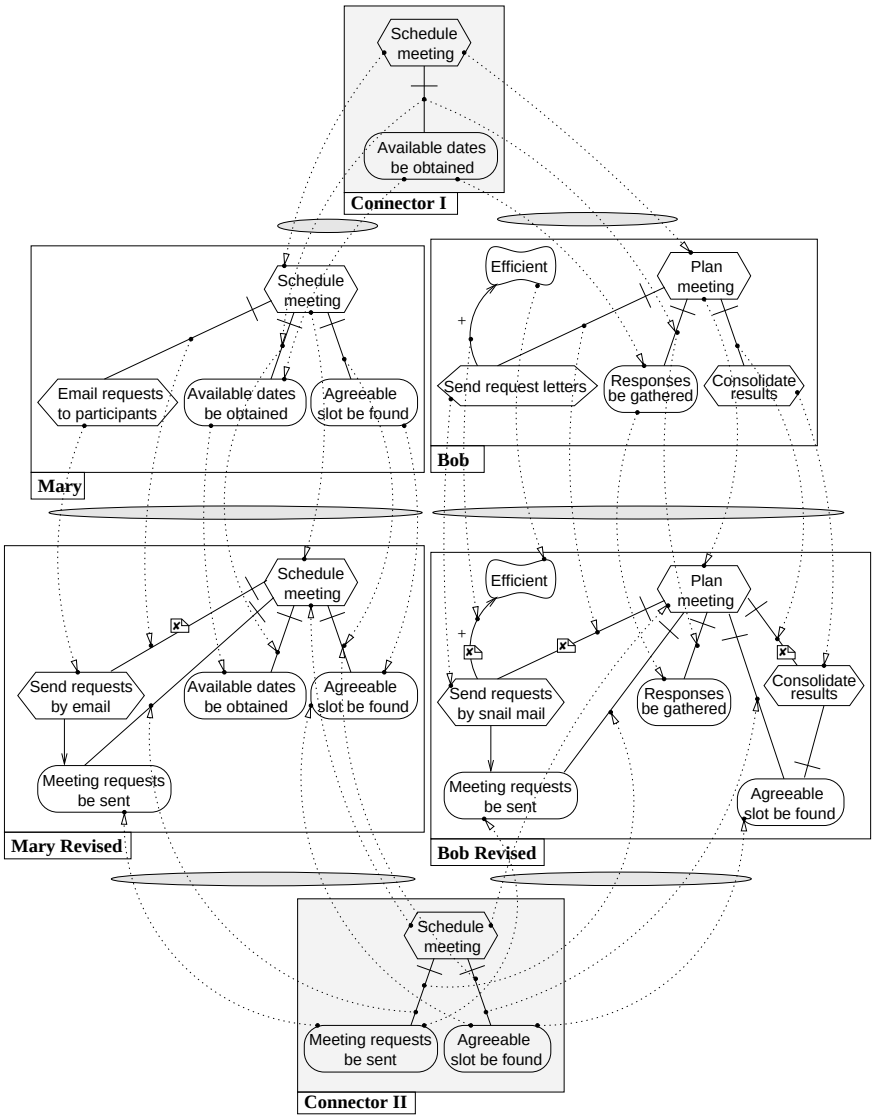


Figure 18: Interconnecting the views

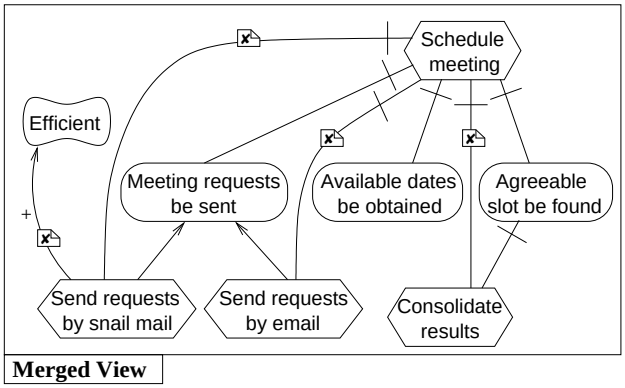


Figure 19: The merged view

the contributions of individual stakeholders. 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 $\big(\{\mathbf{Sam}, \mathbf{Bob}, \mathbf{Mary}\}, \{\mathbf{Sam} \mapsto \lightning, \mathbf{Bob} \mapsto \lightning, \mathbf{Mary} \mapsto \lightning\}\big)$ (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 $\mathcal{M}_{i*}$-typed $(\mathcal{P}, \mathcal{P})$-annotated graphs and use $\mathcal{M}_{i*}$-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 20. We use a concise notation to represent the $\mathcal{P}$-value annotating each element. For example, **B:!;S:✘** means $\big(\{\mathbf{Bob}, \mathbf{Sam}\}, \{\mathbf{Bob} \mapsto \mathbf{!}, \mathbf{Sam} \mapsto \text{✘}\}\big)$ and **M:!;B:!** means $\big(\{\mathbf{Mary}, \mathbf{Bob}\}, \{\mathbf{Mary} \mapsto \mathbf{!}, \mathbf{Bob} \mapsto \mathbf{!}\}\big)$. Note that in the "Connector" view in Figure 20, the elements have no stakeholder annotations, indicated using the bottom element of $\mathcal{P}$, denoted $\emptyset$.

Merging the interconnected views in Figure 20 yields the merged view shown in Figure 21. The annotation for each element of this view reflects the decisions made about the element by the involved stakeholders. The supremum of the annotations on each element in Figure 21 results in the value annotating the corresponding element in Figure 19.

Our example involved only three stakeholders. The annotation universe for n stakeholders $s_1, \ldots, s_n$ is defined similarly: given a (complete) knowledge-ordering lattice $\mathcal{L}$, the annotation universe is the $\mathcal{L}$-annotated powerset induced by $\big(\{s_1, \ldots, s_n\}, \{s_1 \mapsto \top_{\mathcal{L}}, \ldots, s_n \mapsto \top_{\mathcal{L}}\}\big)$ where $\top_{\mathcal{L}}$ denotes the top element of $\mathcal{L}$. If the set of stakeholders is not known beforehand, we can use the $\mathcal{L}$-annotated powerset induced by $\big(\mathbb{N}, \{n \mapsto \top_{\mathcal{L}} \mid n \in \mathbb{N}\}\big)$ where $\mathbb{N}$ is the set of natural numbers. We can then add more stakeholders as needed by *binding* each new stakeholder to a natural number not already taken by another stakeholder.

7 Discussion

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

7.1 Sanity Checks

The typing mechanism discussed in Subsection 3.2 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 formulation involves multiplicities or relies on the semantics of the modeling language. In the class diagram example discussed in Subsection 3.2, 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 [16]. 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.

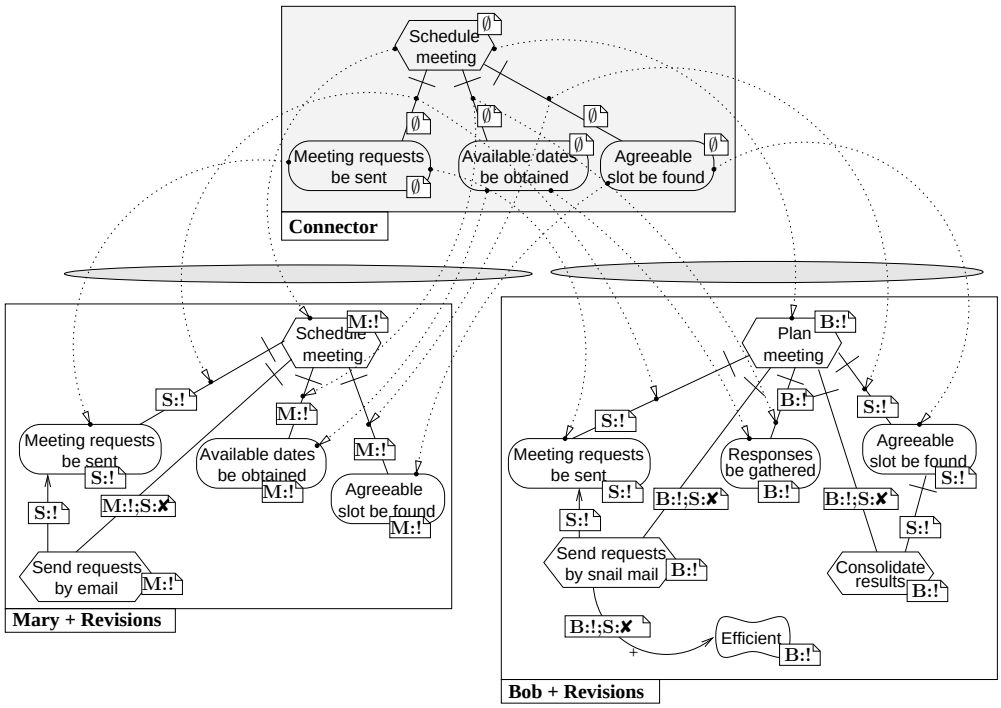


Figure 20: Interconnecting powerset-annotated views

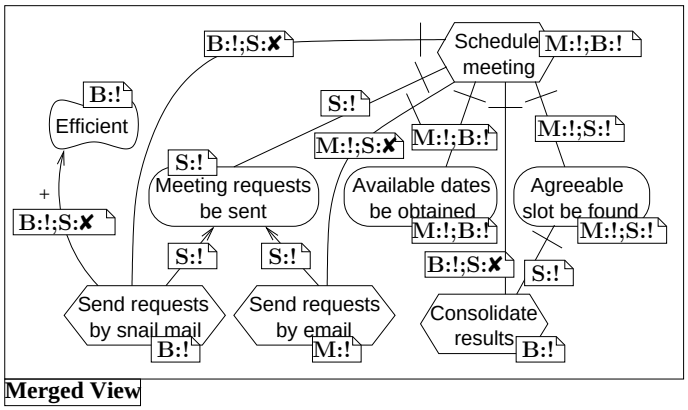


Figure 21: The powerset-annotated merged view

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*. Unfortunately, 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 [25], 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 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 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.

Poset-annotated graphs bear some similarity to fuzzy graphs [17]. What distinguishes our work from the body of work done on fuzzy graphs is our emphasis on algebraic structural relationships rather than graph-

theoretic analysis techniques.

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 well 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* ***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.*

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

[4] M. Chechik et al. Multi-valued symbolic model-checking. *TOSEM*. To appear.

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

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

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

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

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

[10] 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.

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

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

[13] J. Goguen. Concept representation in natural and artificial languages. *Intl. J. of Man-Machine Studies*, 6:513–561, 1974.

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

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

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

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

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

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

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

[21] D. Rydeheard and R. Burstall. *Computational Category Theory*. Prentice Hall, 1988.

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

[23] A. Tarlecki et al. Some fundamental algebraic tools for the semantics of computation, part III. *TCS*, 91:239–264, 1991.

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

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

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