View Merging in the Presence of Incompleteness and Inconsistency

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

Abstract

View merging, also called view integration, is a key problem in conceptual modeling. Large models are often constructed and accessed by manipulating individual views, but it is important to be able to consolidate a set of views to gain a unified perspective, to understand interactions between views, or to perform various types of analysis. View merging is complicated by incompleteness and inconsistency: Stakeholders often have varying degrees of confidence about their statements. Their views capture different but overlapping aspects of a problem, and may have discrepancies over the terminology being used, the concepts being modeled, or how these concepts should be structured. Once views are merged, it is important to be able to trace the elements of the merged view back to their sources and to the merge assumptions related to them. In this paper, we present a framework for merging incomplete and inconsistent graph-based views. We introduce a formalism, called annotated graphs, with a built-in annotation scheme for modeling incompleteness and inconsistency. We show how structure-preserving maps can be employed to express the relationships between disparate views modeled as annotated graphs, and provide a general algorithm for merging views with arbitrary interconnections. We provide a systematic way to generate and represent the traceability information required for tracing the merged view elements back to their sources, and to the merge assumptions giving rise to the elements.

Keywords: Model Management, View-Based Development, View Merging, Inconsistency Management

1 Introduction

Models play a key role in many aspects of requirements analysis and design. Developers build models of the problem domain, to understand the relationships between stakeholders and their goals, and they build models of the system under development, to reason about its structure, behavior, and function. For complex systems, these models are constructed and manipulated by distributed teams, each working on a partial view of the overall system. Building a consistent, unified view is a major challenge [1].

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 their analyses to refine their ideas about the system being modeled. Hence, views may evolve over time. Multiple versions of some views may be created to explore competing alternatives, or to respond to changing requirements. Therefore, most of the time, the current set of views are likely to be incomplete and inconsistent [2].

The term *model management* describes the set of activities concerned with keeping track of the relationships between these views, and managing consistency as they evolve. Bernstein [3] identifies a number of useful operators on views, including Match (to find mappings between models), Diff (to find the differences between models), and Merge (to compute the union of a set of models, according to known mappings between them).

In this paper, we concentrate on view merging. 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 among views, or to perform various types of end-to-end analysis.

A number of approaches for view merging have been proposed [4, 5, 6, 7, 8]. However, these approaches assume the set of views are consistent prior to merging. This is fine if the views were carefully designed to work together; however, for most interesting applications, the views are not likely to be consistent *a priori*. A set of views are inconsistent if some relationship that should hold between them does not hold [9]. In the literature on view merging, the nature of such consistency relationships is often left implicit, and so the problem of inconsistent views is ignored. In general, existing approaches to view merging can only be used if considerable effort is put into detecting and repairing inconsistencies first.

Recent work on inconsistency management tools [10] helps in this respect but does not entirely address the problem because, as we will argue, it is not possible to determine whether a set of views are entirely consistent until all the decisions are made about exactly how they are to be merged. The intended relationships between the views must be stated precisely.

In this paper, we present a framework for merging multiple views that tolerates inconsistency between the views. The framework can be adapted to any graph-based modeling language, as it treats the mappings between views in terms of mappings between nodes and edges in the underlying graphs. We will demonstrate the application of the framework to the early requirements modeling language i^* [11] and to entity-relationship models. In ongoing work we are exploring application to the various modeling notations of UML.

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

We use category theory [12] as a theoretical basis for our merge framework. We treat views as structured objects, and the intended relationships between them as structure-preserving mappings. To model incompleteness and inconsistency, we annotate view elements with labels denoting the amount of knowledge available about them. To ensure proper evolution of annotations, we constrain how these labels can be treated in the mappings that interrelate views. We provide a mathematically rigorous merge algorithm based on an algebraic concept called *colimit*. This treatment offers both scalability to arbitrary numbers of views, and adaptability to different conceptual modeling languages.

After computing a merge, we may need to know how the original views and the defined mappings between them participated in producing the result. Our framework provides the ability to trace the elements of the merged view back to the originating views, to the contributing stakeholders, and to the view interrelationship assumptions related to the elements. We discuss how the information required for addressing each of these traceability concerns can be generated and represented in our framework.

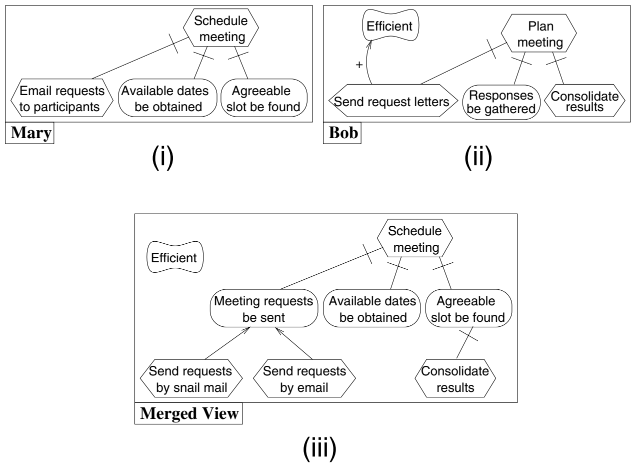


Figure 1. Merging i^* views

Parts of this paper have previously been published in a research paper at the 13th IEEE International Requirements Engineering Conference (RE'05) [13], in a demo paper at the same conference [14], and in a workshop paper at the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE '05) [15]. This paper brings together the ideas described in these earlier papers, to provide a definitive treatment of our merge framework.

2 Motivating Examples

We will use two working examples throughout the paper, one involving goal models represented in the i^* notation, and another involving database schemata captured by entity relationship diagrams. Through these applications, we demonstrate how the ideas presented in this paper can be used for the management of requirements elicitation artifacts, and to support the exploratory view merging process. This section briefly explains these examples, and uses them to illustrate the main challenges in view merging.

2.1 Merging i^* Models

A requirements analyst, Sam, is developing a goal model for a meeting scheduler [16], 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. He then merges the views to study how well their goals fit together.

Figures 1(i) and 1(ii) show the initial views of Mary and Bob. At first sight, there appears to be no overlap, as Mary and Bob use different terminologies. However, Sam suspects there are some straight-forward correspondences:

Schedule meeting in Mary's view is probably the same task as Plan meeting in Bob's. Mary's Available dates be obtained may be the same goal as Bob's Responses be gathered. Sam also thinks it makes sense to treat Mary's Email requests to participants and Bob's Send request letters as alternative ways of satisfying an unstated goal, Meeting requests be sent. Bob's Consolidate results task appears to make sense as a subtask of Mary's Agreeable slot be found goal. Finally, after seeing both views, Mary points out that Bob's positive contribution link from Send request letters to the Efficient soft-goal is inappropriate, although she believes the Efficient soft-goal itself is important.

For a problem of this size, Sam would likely just do an ad-hoc merge with a result such as Figure 1(iii), and show this to Bob and Mary for validation. This (ad-hoc) merge 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 an ad-hoc merge, Sam will naturally tend to repair inconsistencies implicitly and align the stakeholders' views with his own vision of the merge. Hence, we lose the opportunities to analyze inconsistencies that arise with a particular choice of merge.

- We have lost the ability to trace conceptual contributions. If it is important to capture stakeholders' contributions in individual views, then it must be equally important to keep track of how these contributions get adapted into the merged view.

2.2 Merging Entity Relationship Models

In the i^* view merging example in Section 2.1, the merged view (Figure 1(iii)) would most likely turn out to be agreeable to both Bob and Mary. However, in a more realistic elicitation problem, arriving at a viable consolidation is seldom as easy: View merging is an iterative and evolutionary process where stakeholders constantly refine their perspectives as a result of gaining more knowledge about the problem, and looking back at previous merges and studying how their views affect and are affected by other parties' intentions. To illustrate this, consider the following example: Suppose Sam, the analyst, now wants to develop a database schema for a payroll system based on Bob's and Mary's perspectives. Views are described using Entity Relationship Diagrams (ERDs).

After sketching Mary's and Bob's initial views (Figure 2), Sam will merge them to produce a unified schema.

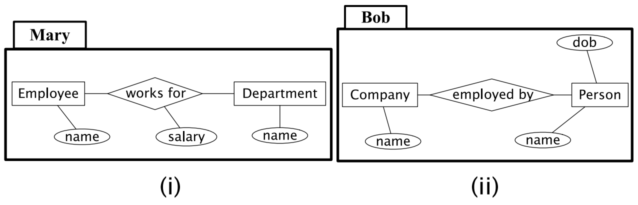


Figure 2. Initial views of stakeholders

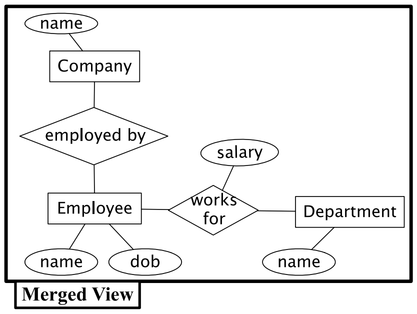


Figure 3. First merge attempt

He identifies the following correspondences between the two views: Employee in Mary's view is likely to be the same entity as Person in Bob's; and consequently, their name attributes are probably the same. Merging Mary's and Bob's views w.r.t. these correspondences results in a schema like the one shown Figure 3. For naming the elements of the merged schema, Sam favored Mary's naming choices over Bob's.

When this merge is presented to Mary, she notices Company, an entity she had not included in her original view. She finds the entity to be important; however, she prefers to call it Corporation. She also decides to add an aggregation link from Corporation to Department. Further, she deems Bob's employed by relationship to be redundant in the light of the works for relationship and the aggregation link from Corporation to Department. The new merged schema addressing Mary's concerns is shown in Figure 4.

When this new schema is shown to Bob, he finds out that the employed by relationship has been dropped from the

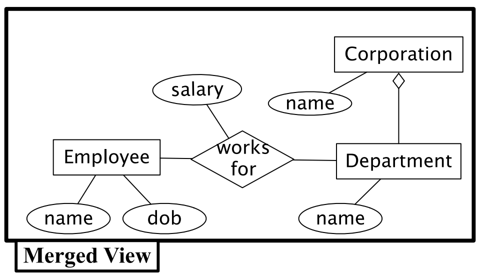


Figure 4. Second merge attempt

merge; however, he argues that there is no redundancy, as it is possible for some employees not to be attached to a particular department. Therefore, he insists that the relationship be added back to the merge!

An ad-hoc merge, or even a structured one computed in a classical framework would fail in at least two respects when faced with a problem such as the one described above:

- It is not possible to describe how sure stakeholders are about elements of their views, and how their beliefs evolve over time. If we later need to know *how flexible* a stakeholder is w.r.t. a certain functionality, we have no way of discovering how strongly the stakeholder argued for (or against) the functionality.

- Disagreements between stakeholders would need to be resolved immediately after being identified because we have no means to model such disagreements explicitly. This is unsatisfactory – previous work suggests that toleration of inconsistencies and disagreements, and being able to delay their resolution is basis for flexible development [9].

The view merging framework that we present in this paper addresses all the problems motivated by the examples in Sections 2.1 and 2.2.

3 View Merging as an Abstract Operation

The view merging framework presented in this paper is based on a category-theoretic concept called *colimit* [12]. In this paper, we will not provide a formal introduction to colimits, but rather will explain the intuitions that motivate our use of category theory. A formal treatment of the main constructs of our approach can be found in [17].

Intuitively, a *category* is an algebraic structure consisting of a collection of *objects* together with a collection of *mappings* (also known as *arrows* or *morphisms*). Each mapping connects a pair of objects, known as its *source* and *destination*. Typically, the objects will have some internal structure, and the mappings express ways in which the structure of one object maps onto that of another. For example, if the objects are geometric shapes, then the mappings could be transformations that preserve shape, such as rotation and scaling. This gives rise to a number of familiar constructs – for example, if a mapping between two objects has an inverse, then we say the two objects are *isomorphic*, i.e. the objects have the same structure.

The appeal of category theory is that it provides a formal foundation for manipulating collections of objects and their mappings. In our case, the objects are views, and the mappings are known or hypothesized relationships between them. We can express a hypothesis about how a group of objects are related using an *interconnection diagram*. An interconnection diagram is a set of objects of a particular category and a subset of possible mappings between them[1].

The *colimit* of an interconnection diagram is a new object, called the *colimiting object*, together with a family of mappings, one from each object in the diagram onto the colimiting object[2]. Since each mapping expresses how the internal structure of its source object is mapped onto that of its destination object, the colimit expresses the merge of all the objects in the interconnection diagram. Furthermore, the colimit respects the mappings in the diagram: The intuition here is that the image of each object in the colimit is the same, no matter which path through the mappings in the diagram you follow. By definition, the colimit is also minimal – it merges the objects in the diagram without adding anything essentially new [18].

To merge a set of views, we first express how they are related in an interconnection diagram, and then compute the colimit. For example, if we want to merge two views, A and B, that overlap in some way, we can express the overlap as a third view, C, with mappings from C to each of A and B:

$$\begin{tikzcd}
A & & B \\
& C \arrow[ul, "f"] \arrow[ur, "g"'] &
\end{tikzcd}$$

In this interconnection diagram, the two mappings f and g specify how the common part, C, is represented in each of A and B. The colimit of this diagram is a new view, P, expressing the union of A and B, such that their overlap, C, is included only once. This simple interconnection pattern is known as a *three-way merge*.

The reason why we hypothesize merges explicitly and define the merge operation in terms of specific interconnection diagrams, rather than in terms of *all* given views and mappings is because, from time to time, we may want to create merges using only a subset of the existing views. We therefore need to be able to specify which views are involved in each merge. Further, we may have several competing versions of mappings between any two participating views making it necessary also to specify which mappings are to be used for computing a particular merge.

In practice, interconnection diagrams often have more complex patterns than that of three-way merge. Figure 5 shows two examples used later in this paper: 5(i) is used for capturing the relationships between the ***i**** meta-model fragments in Figure 10, and 5(ii) is used for capturing the relationships between the views in Figure 13.

[1]The notion of interconnection diagram in category theory is more general than this (cf. e.g. [12]), but the extra generality is unnecessary here.

[2]In the remainder of the paper, with a slight abuse of terminology, we use the term "colimit" to refer to the colimiting object for a given interconnection diagram.

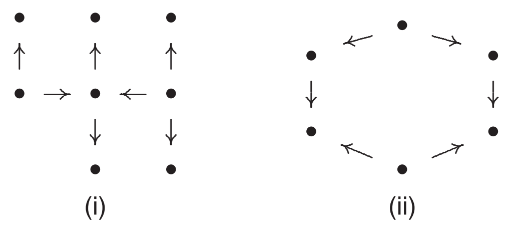


Figure 5. Examples of interconnection patterns

It can be shown that each of the merge algorithms given in this paper corresponds to colimit computation in an appropriate category. Further details about the correspondence between colimits and the algorithms given herein can be found in a technical report [19] where the mathematical underpinnings of our work have been documented.

4 Interconnecting and Merging Graphs

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

4.1 Merging Sets

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

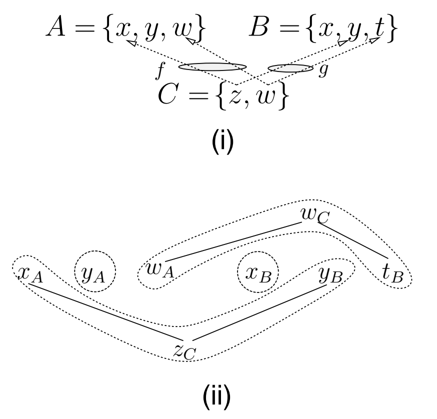
To describe the algorithm for merging sets, we need to introduce the concept of *disjoint union*: The disjoint union of a given family of sets $S_1, S_2, \ldots, S_n$, denoted $S_1 \uplus S_2 \uplus \ldots \uplus S_n$, is (isomorphic to) the following: $S_1 \times \{1\} \cup S_2 \times \{2\} \cup \ldots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

To merge a system of interconnected sets, we start with the disjoint union as the largest possible merged set, and refine it by grouping together elements that get unified by the interconnections. To identify which elements should be unified, we construct a *unification graph* U, a graphical representation of the symmetric binary relation induced on the elements of the disjoint union by the interconnections. We then combine the elements that fall in the same connected component of U. Figure 6 shows the merge algorithm for an interconnection diagram whose objects are sets $S_1, \ldots, S_n$ and whose mappings are functions $f_1, \ldots, f_k$.

Figure 7 shows an example of three-way merge for sets: 7(i) shows the interconnection diagram; 7(ii) shows the induced unification graph and its connected components; and 7(iii) shows the merged set. The example shows that simply taking the union of two sets A and B might not be the right way to merge them as this may cause name-clashes (e.g. according to the interconnections, the y elements in A and B are not the same although they share the same name), or duplicates for equivalent but distinctly-named elements (e.g. according to the interconnections, w in A and t in B are the same despite having distinct names).

```
SET-MERGE ($S_1, \ldots, S_n, f_1, \ldots, f_k$):
  ▷ Let $U$ be an initially discrete graph with node-set $S_1 \uplus \ldots \uplus S_n$;
  ▷ For every function $f_i$   $(1 \leq i \leq k)$:
        ▷ For every element $a$ in the domain of $f_i$:
              ▷ Add to $U$ an undirected edge between the elements
                corresponding to $a$ and $f_i(a)$;
  ▷ Let $P$ be the set of the connected components of $U$;
  ▷ Return $P$ as the result of the merge operation.
```

Figure 6. Algorithm for merging sets



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

(iii)

Figure 7. Three-way merge example for sets

In the above set-merging example, the elements of each set were uniquely identifiable by their names within the set. This is not necessarily the case in general because we may have unnamed or identically-named, but distinct elements. For example, in Sections 2.1 and 2.2, most edges in the views were unnamed; and in Section 2.2, the name node appeared more than once in Bob's and Mary's views as well as the merges. To avoid ambiguity, our implementation of the merge framework (discussed in Section 7) uses *Global Identifiers* (GId's) instead of names to distinguish between view elements.

Name Mapping

To assign a name to each element of the merged set in Figure 7, we combined the names of all the elements in A, B, and C that are mapped to it. For example, "$\{x_A, y_B, z_C\}$" indicates an element that *represents* x of A, y of B, and z of C. A better way to name the elements of the merged set is assigning *naming priorities* to the input sets. For example, in three-way merge, it makes sense to give priority to the element names in the connector, C, and write the merged set in our example as $P = \{z_C, y_A, w_C, x_B\}$. In this particular example, there are no name-clashes in the merged set, so we could drop the element subscripts and write $P = \{z, y, w, x\}$; however, in general, the subscripts are needed to avoid name clashes that arise when stakeholders use the same terms to describe different concepts.

This naming convention is of no theoretical significance, but it provides a natural solution to the name mapping problem: in most cases, we would like the choice of names in *connector objects*, i.e. objects solely used to describe the relationships between other objects, to have precedence in determining the element names in the merged object. We will use this convention in the rest of this paper.

4.2 Graphs and Graph Merging

The notion of graph as introduced below is a specific kind of directed graph used in algebraic approaches to graph-based modeling and transformation [20], and has been successfully applied to capture various graphical formalisms including UML, Entity-Relationship Diagrams, and Petri Nets [21].

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

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

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

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

To determine the source (resp. target) of each edge e in the edge-set of the merged graph P, we pick, among $G_1, \ldots, G_n$, some graph G_i that has an edge q which is represented by e. Let s (resp. t) denote the source (resp. target) of q in G_i; and let s' (resp. t') denote the node that represents s (resp. t) in the node-set of P. We set the source (resp. target) of e in P to s' (resp. t'). Notice that an edge in the merged graph may represent edges from several input graphs. In a category-theoretic setting, it can be shown that the source and the target of each edge in the merged graph are uniquely determined irrespective of which G_i we pick [22].

Figure 8 shows an example of three-way merge for graphs. In the figure, each homomorphism has been visualized by a set of directed dashed lines. In addition to the homomorphisms of the interconnection diagram, i.e. f and g, we have shown the homomorphisms δ_A and δ_B specifying how A and B are represented in P. The homomorphism from C to P is implied and has not been shown.

To compute the graph P in Figure 8, we first separately merged the node-sets and the edge-sets of A, B, C. That is, we merged sets $\{x_1, x_2, x_3\}, \{n_1, n_2, n_3\}, \{u_1, u_2\}$ w.r.t. to functions $f_{\mathbf{node}} = \{u_1 \mapsto x_1, u_2 \mapsto x_2\}, g_{\mathbf{node}} = \{u_1 \mapsto n_1, u_2 \mapsto n_2\}$; and merged $\{p_1, p_2\}, \{e_1, e_2, e_3\}, \{v_1\}$ w.r.t. to $f_{\mathbf{edge}} = \{v_1 \mapsto p_1\}, g_{\mathbf{edge}} = \{v_1 \mapsto e_1\}$. This yielded two sets $N = \{u_1, u_2, x_3, n_3\}$, $E = \{v_1, p_2, e_2, e_3\}$ constituting the node-set and the edge-set of P respectively. For naming the elements of N and E, we gave priority to the choice of names used in graph C (name mapping was already discussed in Section 4.1). After computing N and E, we assigned to each edge in E a source and a target node from N using the method described earlier. We illustrate this with two examples: 1) To determine the source and target of v_1 in E, we need to pick, among A, B, C, a graph that has an edge represented by v_1. In this case, any of the three

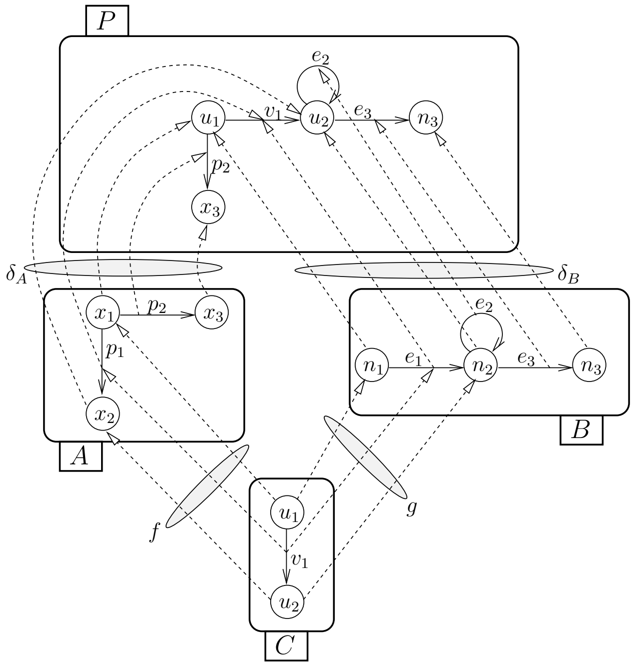


Figure 8. Three-way merge example for graphs

graphs will do because v_1 has a pre-image in each of them – the edge represents p_1 of A, e_1 of B, and v_1 of C. Regardless of which graph we pick, the computed source and target will be the same. Suppose we pick A. Edge p_1 has x_1 as source and x_2 as target. The two nodes are represented in N by u_1 and u_2 respectively; therefore, v_1 is assigned u_1 as source and u_2 as target. 2) Now, consider e_3 in E. The edge has a pre-image in graph B only. Thus, we pick B. Edge e_3 in B has n_2 as source and n_3 as target. Nodes n_2 and n_3 are respectively represented by u_2 and n_3 in N. Thus, e_3 in E is assigned u_2 as source and n_3 as target.

Enforcement of Types

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

In [23], a powerful typing mechanism for graphs has been proposed using the relation between the models and the meta-model for the language. Assuming that the meta-model for the language of interest is given by a graph $\mathcal{M}$, every model is described by a pair $\langle G, t : G \to \mathcal{M}\rangle$ where G is a graph and t is a homomorphism, called the *typing map*, assigning a type to every element in G. Notice that a typing map is a homomorphism, offering more structure than an arbitrary pair of functions assigning types to nodes

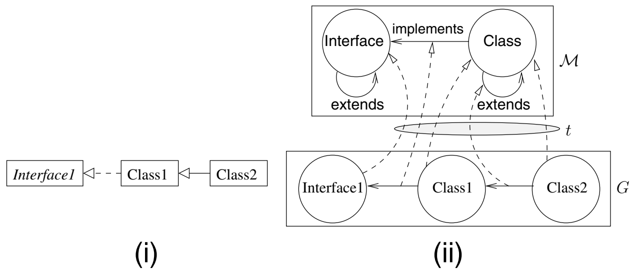


Figure 9. Example of typed graphs

and edges. A typed homomorphism $\underline{h} : \langle G, t\rangle \to \langle G', t'\rangle$ is simply a homomorphism $h : G \to G'$ that preserves types, i.e. $t'(h(x)) = t(x)$ for every element x in G. This typing mechanism is illustrated in Figure 9: 9(i) shows a Java class diagram in UML notation and 9(ii) shows how it can be represented using a typed graph. The graph $\mathcal{M}$ in 9(ii) is the extends–implements fragment of the meta-model for Java class diagrams.

The meta-model for a graph-based language can be much more complex than that of Figure 9. Figure 10 shows some fragments of the $\boldsymbol{i}^*$ meta-model extracted from the visual syntax description of $\boldsymbol{i}^*$'s successor GRL [24]. Instead of showing the whole meta-model in one graph, we have broken it into a number of *views*, each of which represents a particular type of relationship (means-ends, decomposition, etc.). Our graph merging framework allows us to describe the meta-model without having to show it monolithically: the $\boldsymbol{i}^*$ meta-model, $\mathcal{M}_{\boldsymbol{i}^*}$, 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 gray) have been used. Each mapping (shown by a thick solid line) is a homomorphism giving the obvious mapping. Notice that the connector graphs are *discrete* (i.e. do not have any edges) as no two meta-model fragments share common edges of the same type. The $\wedge$- and $\vee$-contribution structures in $\boldsymbol{i}^*$ convey a relationship between a group of edges. To capture this, we introduced helper nodes (shown as small rectangular boxes) in the meta-model to group edges that should be related by $\wedge$ or $\vee$. Figure 11(i) shows how we normally draw an $\vee$-contribution structure in an $\boldsymbol{i}^*$ model and Figure 11(ii) shows the adaptation of the structure to typed graphs. Structures conveying relationships between a combination of nodes and edges can be modeled similarly.

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

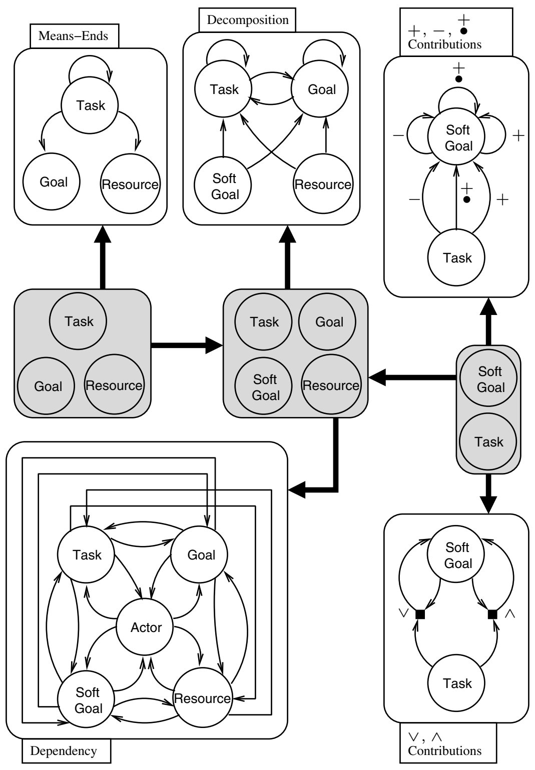


Figure 10. Some meta-model fragments of i^*

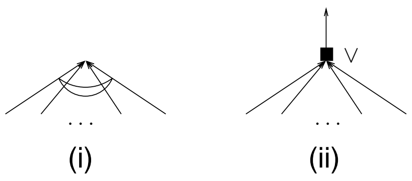


Figure 11. Adaptation of ∨-contribution

tablished from the merged graph to the meta-model [23].

5 Merging Requirements Views

The merge framework of the previous section provides sufficient machinery for merging graph-based views that are free from incompleteness and inconsistency. However, as we argued earlier, for most interesting applications that involve multiple stakeholders, the views are unlikely to be either conclusive or consistent. Therefore, it is crucial to be able to tolerate incompleteness and inconsistency. In this section, we show how incompleteness and inconsistency can be modeled by an appropriate choice of *annotation* for view elements. Using the motivating examples in Section 2, we demonstrate how incomplete and inconsistent views can be represented, interconnected, and merged.

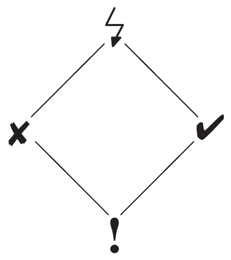


Figure 12. Belnap's knowledge order variant

5.1 Annotated Views

The classical approach discussed in Section 4 is capable of describing view correspondences; however, it provides no means to express the stakeholders' beliefs about the *fitness* of view elements, and the possible ways in which these beliefs can *evolve*. Consequently, we cannot describe *how sure* stakeholders are about each of the decisions they make. Further, we cannot express inconsistencies and disagreements that arise due to discrepancies between stakeholders' decisions about either the structure or the contents of views.

To model stakeholders' beliefs, we attach to each view element an annotation denoting the *degree of knowledge* available about the element. We formalize knowledge degrees using *knowledge orders*. A knowledge order is a partially ordered set [25] specifying the different levels of knowledge that can be associated to view elements, and the possible ways in which this knowledge can grow. The idea of knowledge orders was first introduced by Belnap [26], and later generalized by Ginsberg [27].

One of the simplest and most useful knowledge orders is Belnap's four-valued knowledge order [26]. The knowledge order $\mathcal{K}$ shown in Figure 12 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 conflict as to whether the element is well-conceived, i.e. the element is *disputed* [3].

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

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

[3] Belnap's original lattice refers to **!** as maybe, ✘ as false, ✔ as true, and ϟ as disagreement.

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

Definition 5.2 (annotation-respecting homomorphism) Let $\mathbf{G}$ and $\mathbf{G}'$ be K-annotated graphs. A *K-respecting homomorphism* $\mathbf{h} : \mathbf{G} \to \mathbf{G}'$ is a homomorphism subject to the following condition: For every element (i.e. node or edge) x in $\mathbf{G}$, the image of x under $\mathbf{h}$ has an annotation which is *larger than or equal to* the annotation of x.

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

For a fixed knowledge order K, the merge operation over an interconnection diagram whose objects $\mathbf{G}_1, \ldots, \mathbf{G}_n$ are K-annotated graphs and whose mappings $\mathbf{h}_1, \ldots, \mathbf{h}_k$ are K-respecting homomorphisms, yields a merged object $\mathbf{P}$ computed as follows: First, disregard the annotations of $\mathbf{G}_1, \ldots, \mathbf{G}_n$ and merge the resulting graphs w.r.t. $\mathbf{h}_1, \ldots, \mathbf{h}_k$ to get a graph P. Then, to construct $\mathbf{P}$, attach an annotation to every element x in P by taking the *least upper bound* [25] of the annotations of all the elements that x represents.

Intuitively, the least upper bound of a set of knowledge degrees $S \subseteq K$ is the least specific knowledge degree that refines (i.e. is more specific than) all the members of S. To ensure that the least upper bound exists for any subset of K, we assume K to be a *complete lattice* [25]. The knowledge order $\mathcal{K}$ in Figure 12 is an example of a complete lattice.

As an example, suppose the graphs in Figure 8 were annotated with $\mathcal{K}$ in such a way that the homomorphisms f and g satisfied the condition in Definition 5.2. Assuming that the nodes u_1 of C, x_1 of A, n_1 of B are respectively annotated with **!**, ✔, and ✘, the annotation for the node u_1 of P, which represents the aforementioned three nodes, is calculated by taking the least upper bound of the set $S = \{!, ✔, ✘\}$ resulting in the value ϟ.

Incorporating types into annotated graphs is independent of the annotations and is done in exactly the same manner as described in Section 4. In [19], we provide a complete version of the definitions for typed annotated graphs.

5.2 Example I: Merging i^* Views

We can now demonstrate how to merge the i^* views of Figure 1. We assume views are typed using the i^* metamodel $\mathcal{M}_{i^*}$ (cf. Section 4), and will use the lattice $\mathcal{K}$ (Figure 12) for annotating view elements. We therefore express relationships between views by ($\mathcal{M}_{i^*}$-typed) $\mathcal{K}$-respecting homomorphisms. Figure 13 depicts one way to express the relationships between the views in Figures 1(i) and 1(ii).

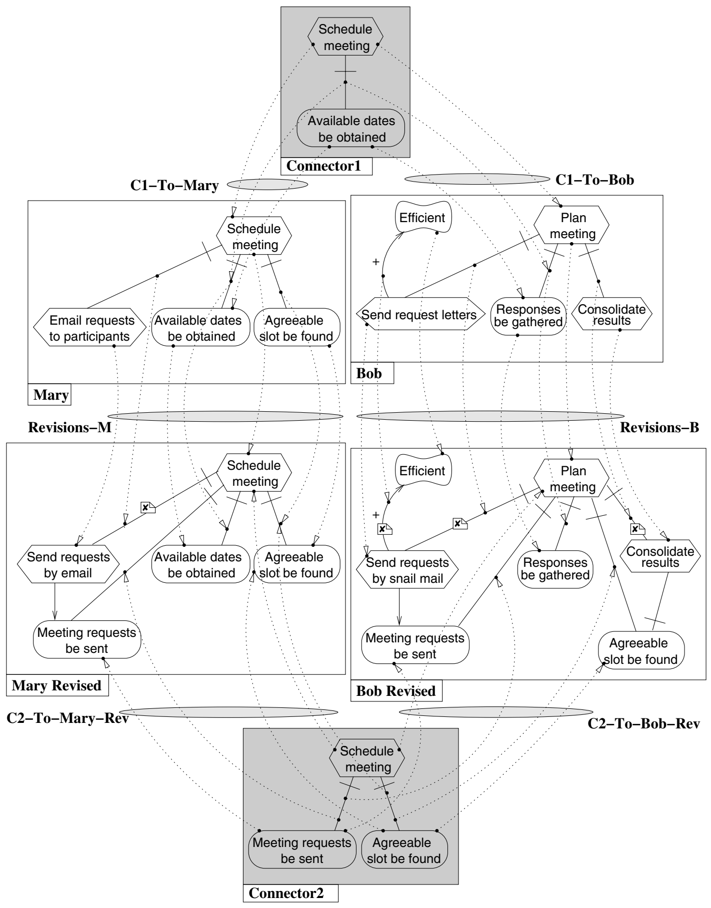


Figure 13. i^* example: Interconnections

For convenience, we treat 'proposed' (**!**) as a default annotation for all nodes and edges, and only show annotations for the remaining values. For example, some edges in the revised versions of Bob's and Mary's views are annotated with ✘ to indicate they are repudiated.

The interconnections in Figure 13 were arrived at as follows. First, Sam creates a connector view **Connector1** to identify synonymous elements in Bob's and Mary's views. Notice that even if Bob and Mary happened to use the same terminology in their views, defining a connector would still be necessary because our merge framework does not rely on naming conventions to describe the desired unifications – all correspondences must be identified explicitly prior to the merge operation.

To build **Connector1**, Sam merely needs to declare which nodes in the two views are equivalent. Because i^* does not allow parallel edges of the same type between any pair of nodes, the edge interconnections are identified automatically once the node interconnections are declared. For example, when Mary's Schedule meeting and Available dates be obtained are respectively unified with Bob's Plan meeting

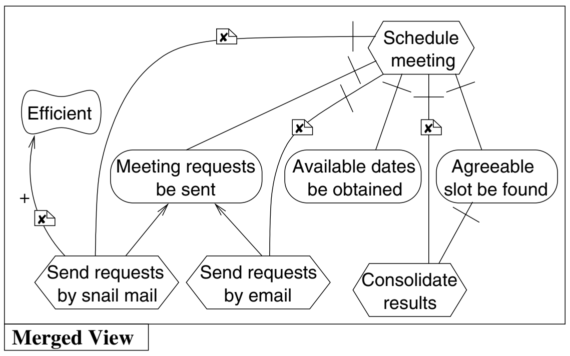


Figure 14. i^* example: The merged view

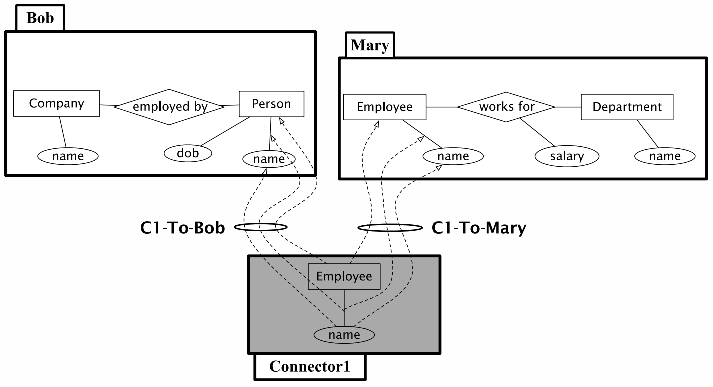


Figure 15. ER example: Interconnections (Part I)

and Responses be gathered, the decomposition links between them in the two views should also be unified.

Next, Sam elaborates each of Bob's and Mary's views to obtain **Mary Revised** and **Bob Revised**. In these views, Sam has repudiated the elements he wants to replace, and proposed additional elements that he needs to complete the merge. Sam could, of course, affirm all the remaining elements of the original views, but he preferred not to do so because the models are in very early stages of elicitation. Finally, Sam keeps track of cases where the same element was added to more than one view using another connector view, **Connector2**.

With these interconnections, the views in Figure 13 can be automatically merged, to obtain the view shown in Figure 14. To name the elements of the merged view priority has been given to Sam's choice of names. For presentation, we may want to *mask* the elements annotated with ✘. This would result in the view shown in Figure 1(iii).

In the above scenario, we treated the original elements of Mary's and Bob's views as being at the *proposed* level, allowing further decisions to be freely made 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 merge hypothesis with the appropriate mappings from Mary's or Bob's original views. When we recompute the merge, the new annotations may result in disagreements. We illustrate this in Section 5.3.

5.3 Example II: Merging ER Views

To merge the ER views of Figure 2, we assume them to be typed by a meta-model $\mathcal{M}_{\mathbf{ER}}$. We chose to omit this meta-model in the paper because the process of constructing it is similar to that described in Section 4 for constructing $\mathcal{M}_{i^*}$. As in the previous example, the lattice $\mathcal{K}$ (Figure 12) will be used to annotate view elements, and 'proposed' (!) will be treated as a default annotation. Relationships between views will be expressed by $\mathcal{K}$-respecting homomorphisms.

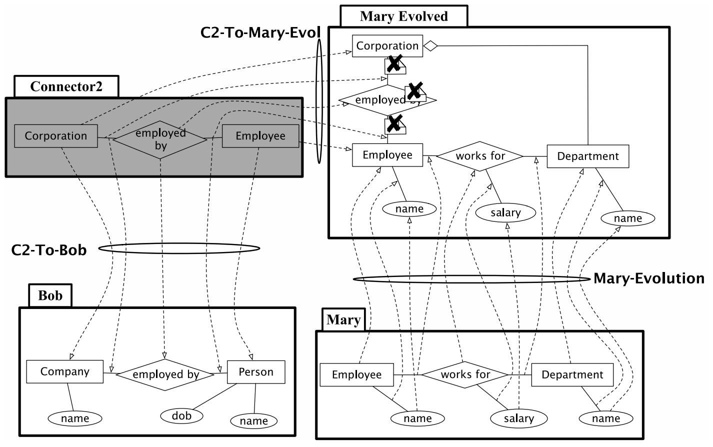


Figure 16. ER example: Interconnections (Part II)

First attempt: In the first iteration, Sam describes the correspondences between Bob's and Mary's views using a connector view, **Connector1** (Figure 15), and two mappings **C1-To-Bob** and **C1-To-Mary**. Merging the interconnection diagram made up of views **Bob**, **Mary** and **Connector1**, and mappings **C1-To-Bob** and **C1-To-Mary** yields the schema shown in Figure 3.

Second attempt: In the second iteration, Mary *evolves* her original view, to obtain **Mary Evolved** (Figure 16), addressing the concerns that occurred to her after the first merge attempt. With Sam's help, she establishes the required interrelationships between her evolved view and Bob's original view through a new connector, **Connector2** (Figure 16). If

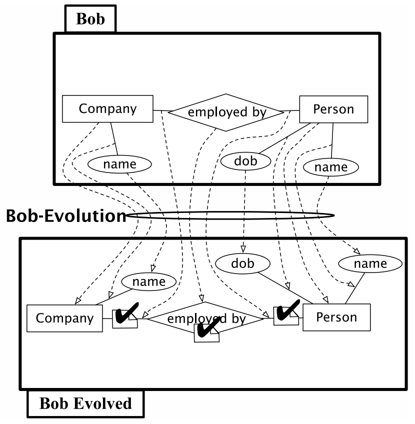


Figure 17. ER example: Interconnections (Part III)

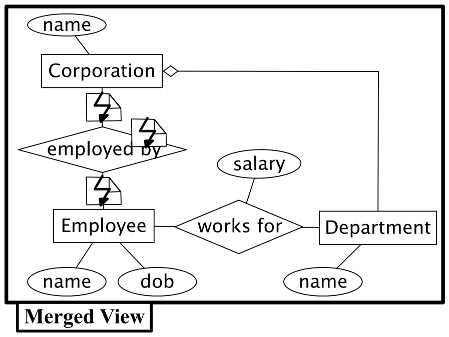


Figure 18. ER example: The merged view

we add these new views and mappings to the interconnection diagram for the first merge attempt and then recompute the merge, we get a schema (not shown) in which the employed by relationship has been repudiated and an aggregation link has been introduced between Corporation and Department. Masking the repudiated elements of this merge will give us the schema in Figure 4.

Third attempt: Finally, in the last iteration, Bob evolves his view (Figure 17), capturing his refined beliefs about the employed by relationship. Adding this new view and mapping leads to the full interconnection diagram shown in Figure 24. Merging according to this interconnection diagram yields the schema shown in Figure 18. In this schema, the annotation computed for the employed by relationship is 'disputed' (ϟ) because Bob and Mary have respectively affirmed and repudiated the corresponding elements in their evolved views.

Since we now have a placeholder in the merge to represent the disagreement between Bob and Mary, there is no need for an immediate resolution – we can delay resolving the conflict, or if it later turns out that the problem is unimportant, we may even elect to ignore it all together.

6 Support for Traceability

After a merge is completed, it is often desirable to know how the input artifacts to the merge process, namely views and mappings, influenced the result. Particularly, it is important to be able to trace the elements of the merged view back to the originating views, and to track the correspondence assumptions behind each unification. We call the former notion *origin traceability* and the latter *assumption traceability* [15]. A third traceability notion, which we refer to as *stakeholder traceability*, arises when multiple stakeholders are allowed to work on individual views. To be able to trace decisions back to their human sources in this setting, we need to differentiate between the conceptual contributions of different stakeholders in individual views.

6.1 Origin and Assumption Traceability

The merges in Section 5 lack the traceability information required for determining where each of their elements originate from. To keep track of the origins of the elements in a merge, the merge operator must store proper traceability links in the merged view.

It turns out that unification graphs, as introduced in Section 4.1, immediately provide the information needed for supporting origin traceability: For a given merge problem, the set of nodes in each connected component of the unification graph constitutes the origin information for the corresponding merged element. For example, the Available dates be obtained goal in Figure 14 should be traceable to Available dates be obtained in **Connector1**, **Mary** and **Mary Revised**, as well as to Responses be gathered in **Bob** and **Bob Revised**.

To trace the correspondence assumptions involved in a unification, we need to know the details of the interrelations among the input view elements that are unified to form an element of the merged view. In a simple scenario such as the first merge attempt in Section 5.3, identifying the correspondence assumptions is trivial because all these assumptions are localized to the mappings **C1-To-Mary** and **C1-To-Bob**. Therefore, if we later need to check why, for example, Person in **Bob** was unified with Employee in **Mary**, we can easily find the chain of correspondences that brought about the unification: Person(**Bob**) = Employee(**Connector1**) by **C1-To-Bob**, and Employee(**Connector1**) = Employee(**Mary**) by **C1-To-Mary**.

However, as merge scenarios get more complex, finding the correspondence assumptions becomes harder. As an example, consider the interconnection diagrams in Figures 15-17: the assumptions about correspondences between views are scattered among several mappings. For example, the

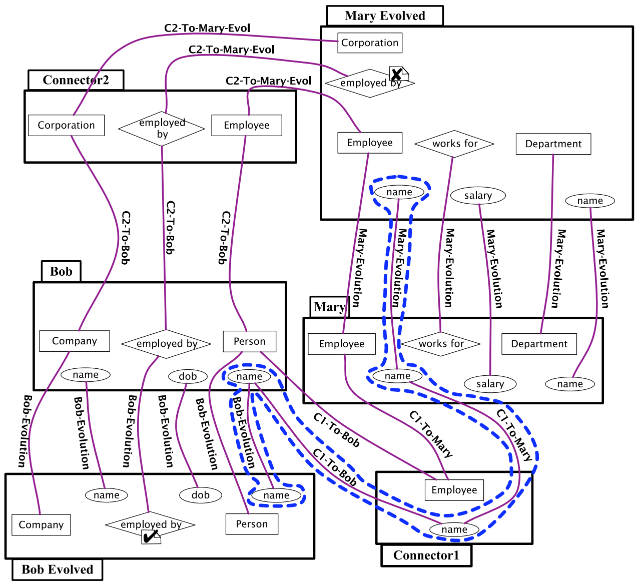


Figure 19. Extended unification graph for the node-sets in Figures 15-17

unification of Company in **Bob Evolved** and Corporation in **Mary Evolved** involves **Bob-Evolution**, **C2-To-Bob** and **C2-To-Mary-Evol**; and the unification of Person's name attribute in **Bob Evolved** and Employee's name attribute in **Mary Evolved** involves **Bob-Evolution**, **C1-To-Bob**, **C1-To-Mary** and **Mary-Evolution**. More interestingly, the unification of Employee in **Bob Evolved** and Employee in **Mary Evolved** can be traced to two different correspondence chains, one involving **Bob-Evolution**, **C1-To-Bob**, **C1-To-Mary** and **Mary-Evolution**; and another involving **Bob-Evolution**, **C2-To-Bob** and **C2-To-Mary-Evol**.

The current notion of unification graph is not readily applicable to finding the correspondence chain(s) involved in creating the elements of the merged view. This is because we do not keep track of *which* mapping induces each of the edges in a unification graph. To address this, we label each edge in a unification graph with the name of the mapping that induces the edge. Figure 19 shows the extended unification graph for merging the node-sets of the views in Figures 15-17. Each connected component of this graph corresponds to one node in the merged view shown in Figure 18. As an example, we have explicitly shown in Figure 19 the connected component corresponding to the Employee's name attribute.

To support both origin and assumption traceability, we store in each element of the merged view a reference to the corresponding connected component of the extended unification graph. Figures 20(i)–(iii) respectively show the stored traceability information for three representative el-

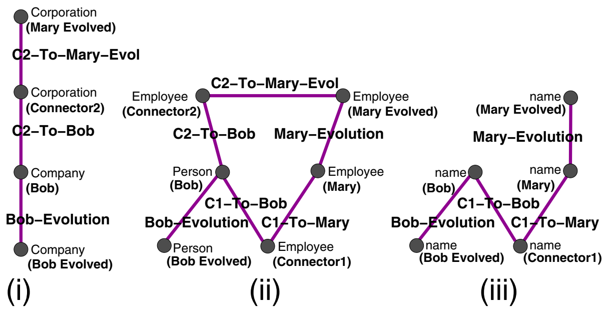


Figure 20. Examples of traceability links

ements of the merged view in Figure 18: Corporation, Employee, and Employee's name. In each case, the traceability information makes it possible to trace the respective element back to its origins, and to the related assumptions in the unification. If we want to see why, for example, Employee in **Mary Evolved** was unified with Person in **Bob Evolved**, we find the *(non-looping) paths* between Employee (**Mary Evolved**) and Person (**Bob**) in Figure 20(ii).

To avoid clutter, we chose not to show the element GId's in Figure 20; however, we should emphasize that GId's need to be kept in the traceability links in order to avoid ambiguity, because an element may not be uniquely identifiable by its name.

6.2 Stakeholder Traceability

When collaborative work is allowed on an individual view, we can no longer assume that all contributions in a given view come from a single human source. The framework developed in Section 5 does not support collaborative work on views because the knowledge labels do not indicate *whose* knowledge is being captured; therefore, we have to assume all contributions in a given view belong to a single stakeholder.

To support tracing contributions back to their human sources when individual views are collaboratively developed, we introduce a more elaborate annotation scheme: Rather than annotating view elements with single annotations, we attach an annotation-set to each element. Each annotation in the annotation-set has a qualifier denoting the stakeholder whose belief is captured by that annotation.

To ensure that the annotation-set X_e attached to a view element e evolves sanely along a mapping $\mathbf{h}$, the following condition must hold: Every stakeholder who has an annotation in X_e must have an annotation in the annotation-set of e's image under $\mathbf{h}$, and this annotation must be at least as specific as that in X_e. Notice that this condition does not prevent $\mathbf{h}$ from introducing annotations for stakeholders who do not already have an annotation in X_e – what is required is that the evolution of already-existing annotations along $\mathbf{h}$ must respect the knowledge order.

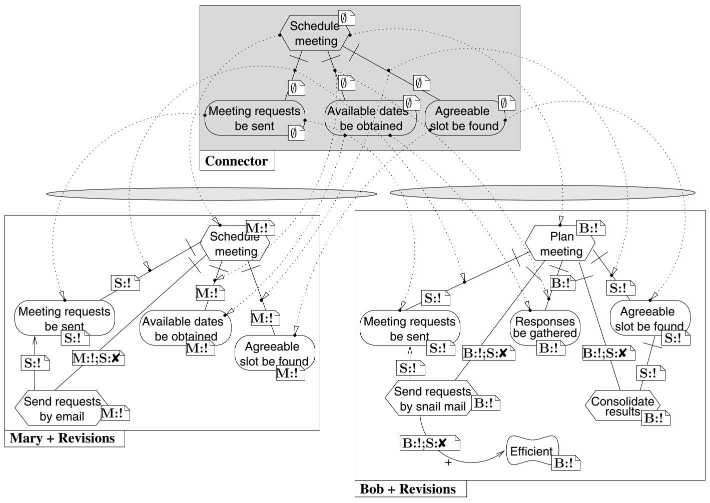


Figure 21. New view interconnections

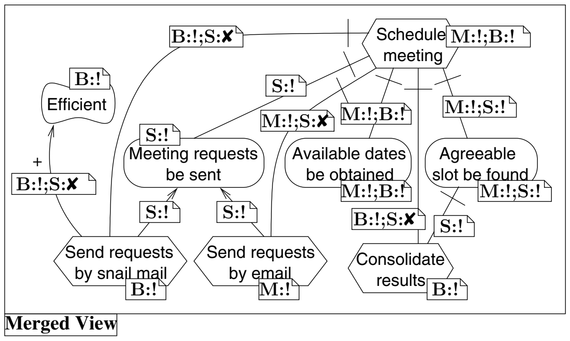


Figure 22. Merged view with detailed annotations

To illustrate the new annotation scheme consider the ***i**** merging example: Sam, Mary, and Bob can now manipulate each others' views without compromising traceability. This is because the annotations can keep track of the contributions of individual parties. The new system of interconnected views is shown in Figure 21. We use a concise notation to represent the annotation-set for each element. For example, **M:!;B:!** means that both Mary and Bob proposed the element; **B:!;S:✘** means Bob proposed the element and Sam repudiated it. Note that in the **Connector** view in Figure 21, the elements have no stakeholder annotations, indicated using $\emptyset$. If we were interested in tracking the revisions Sam makes to Bob's and Mary's vocabularies, we would need to use the same interconnection pattern as that in Figure 13, but the view elements would be annotated with annotation-sets instead of single knowledge degrees.

Merging the interconnected views in Figure 21 yields the view shown in Figure 22. The annotation-set for each view element e in the figure is computed by first unioning the stakeholders that have contributed to the elements represented by e; and then, for every stakeholder s in the union, taking the least upper bound of s's contributions to these elements. In [19], we provide a lattice-theoretic characterization of the computation of annotation-sets using the concept of *annotated powerset lattices*.

The annotation for each element of the merged view in Figure 22 reflects the decisions made about the element by the involved stakeholders. Note that when stakeholders do not work on input views collaboratively, origin traceability subsumes stakeholder traceability; that is, we can produce merges with annotations like those in Figure 22 based on origin traceability information.

7 Tool Support

We have implemented a Java tool, ***i**VuBlender* [14], for merging requirements views. The tool consists of two components: (1) a generic merge library for annotated graphs; and (2) a Swing-based front-end that allows users to graphically express their views, specify the view relationships, and hypothesize and compute merges. We have used the merge library for merging ***i**** models, ERDs, and state-machines. Currently, the front-end only supports ERDs and simple state-machines, and is restricted to just the knowledge order in Figure 12 for annotating view elements. For future versions, we plan to develop a larger collection of graphical widgets to capture the visual syntax of richer modeling formalisms such as ***i****, and provide support for defining arbitrary knowledge orders.

***i**VuBlender* is not bound to a particular process model for merging. Views, mappings, and interconnection diagrams can be conceived and declared in any order with the obvious constraint that a mapping cannot be specified fully unless its source and target views have been fully elaborated. Since our tool can work with partial mappings, this constraint does not mean that a mapping cannot be specified piece-by-piece or that it cannot co-evolve with its two ends. In fact, it is only when we want to compute the merge of an interconnection diagram that it becomes necessary to ensure the mappings referenced in the diagram are fully specified.

Although ***i**VuBlender* may be used in many different ways for merging views, we have found the process model in Figure 23 to be very effective for exploratory modeling problems. In such problems, we usually start by defining a set of disparate views capturing the initial perspectives of the stakeholders. We then declare a merge hypothesis using an interconnection diagram. The hypothesis may reference a number of connector views for describing the correspondences between the stakeholders' views. We define these connectors and follow it by specifying the mappings referenced in the hypothesis. When we compute the merge for the hypothesis, we get a view combining the set of views referenced in the hypothesis w.r.t. their (hypothesized) re-

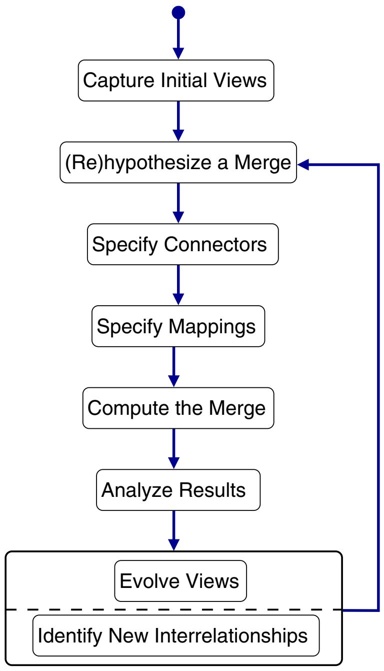


Figure 23. An exploratory merge process

lationships. We may analyze this merged view in various ways. Our analyses often trigger the evolution of views and lead to the discovery of new interrelationships. To incorporate these refinements, we initiate a new iteration of the process by first revising our existing merge hypothesis (or creating a new one) and following the subsequent activities.

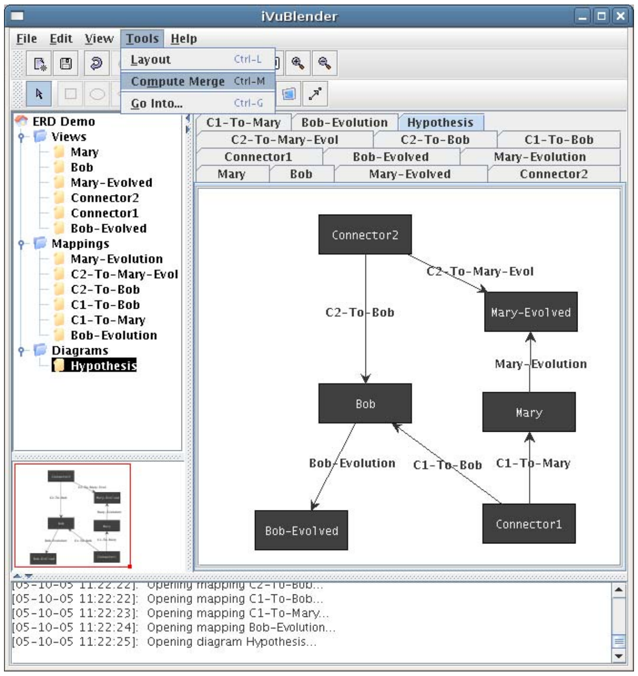
Figures 24 and 25 show two screenshots of *iVuBlender*: The former is a screenshot of the interconnection diagram from Figures 15-17, and the latter is a screenshot of the merged view in Figure 18. To ensure that computed merges are laid out properly, the tool applies a fully automatic layout algorithm to them before presenting them to the user. To represent the annotations, we have used a color scheme in the tool: The default color represents proposed (!), blue and magenta respectively represent affirmed (✔) and repudiated (✘), and red represents disputed (ϟ).

8 Discussion

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

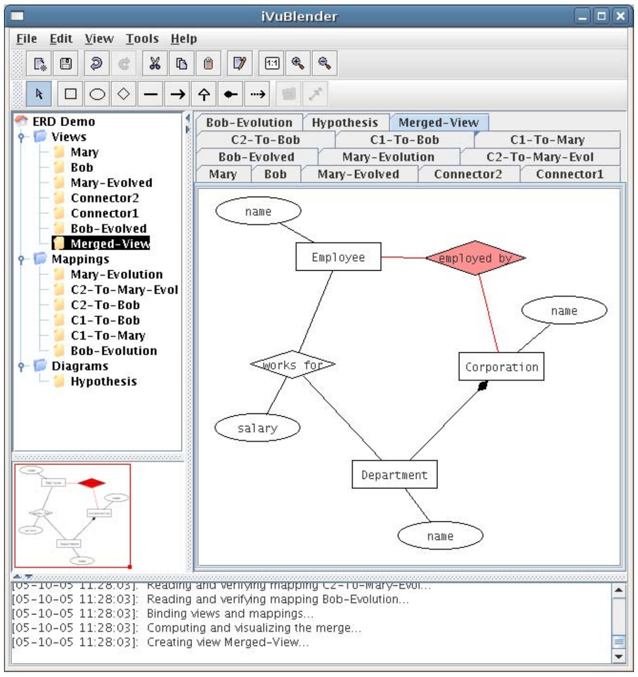
8.1 Sanity Checks

The typing mechanism discussed in Section 4 can capture many classes of constraints which we may need to enforce; however, it has some limitations. Most notably, it cannot capture constraints whose articulation involves the



```
[05-10-05 11:22:22]: Opening mapping C2-To-Bob...
[05-10-05 11:22:22]: Opening mapping C1-To-Bob...
[05-10-05 11:22:23]: Opening mapping C1-To-Mary...
[05-10-05 11:22:24]: Opening mapping Bob-Evolution...
[05-10-05 11:22:25]: Opening diagram Hypothesis...
```

Figure 24. Screenshot of an interconnection diagram



```
[05-10-05 11:28:03]: Reading and verifying mapping C2-To-Mary-Evol...
[05-10-05 11:28:03]: Reading and verifying mapping Bob-Evolution...
[05-10-05 11:28:03]: Binding views and mappings...
[05-10-05 11:28:03]: Computing and visualizing the merge...
[05-10-05 11:28:03]: Creating view Merged-View...
```

Figure 25. Screenshot of a (merged) view

semantics of the modeling language being used. In the class diagram example in Figure 9, we could not express the constraint that a Java class is not allowed to 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.

Since such constraints cannot be expressed in our framework, a number of sanity checks may be needed both on the input views and on the merge to ensure their soundness w.r.t. a desired set of semantics. Even if the input views are sound w.r.t. these semantics, this does not imply that the merge is sound too, because the interconnections do not necessarily preserve semantics. Semantics-dependent sanity checks are formalism-specific. An exposition of such checks for database schema merging can be found in [28].

Another facet to sanity checks in our framework is detecting the potential anomalies caused by annotations. For example, in Section 5, it was possible for a view to have a non-repudiated edge with a repudiated end. In such a case, we would be left with dangling edges if we mask repudiated elements. The detection of such anomalies depends on the interpretation of the annotations being used.

8.2 Identification of Interconnections

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

To our knowledge, little work has been done in Requirements Engineering on automating the identification of view interconnections, even in cases where inconsistency has not been an issue. However, the subject has attracted considerable attention in the Database community for exploring relationships between schemata. There, the identification of interconnections is referred to as *schema matching* [29, 30, 31]. 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 [32], state-machines, and UML to investigate how schema matching techniques can be generalized to graph-based structures other than ERDs.

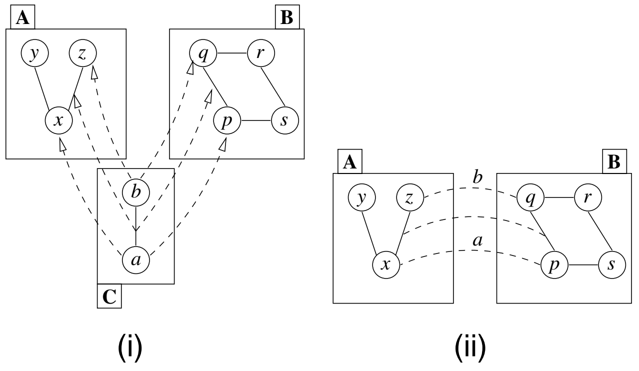


Figure 26. Connectors vs. Direct Mappings

8.3 Specifying Interconnections: Connectors vs. Direct Mappings

In our framework, correspondences between independent views are captured using explicit connector views. For example, to describe the overlap of a pair of views A and B, we create a connector view C that captures just the common parts between A and B, and then specify how C is represented in each of the two views using the mappings $C \to A$ and $C \to B$, as illustrated in Figure 26(i).

Our use of connector views was motivated by theoretical concerns. Briefly, this approach allows us to build the mappings between views using graph homomorphisms – each mapping shows how one view is embedded in another. This in turn allows us to treat views and view mappings as a category, giving us a straightforward way to construct view merges (as colimits), along with standard proofs (e.g. the proof that for this category, the merge always exists and is unique for any set of interconnected views).

One could argue that this approach is less appealing than specifying correspondences between A and B by linking their shared elements directly. Such a scheme could handle naming preferences by encoding the names in labeled binary relations, as illustrated in Figure 26(ii). Note that the mapping in this example is *not* a graph homomorphism.

It would be possible to hide the use of connector views from the user in ***iVuBlender*** by presenting them as direct links instead. However, the use of explicit connector views offers several methodological advantages:

- It allows us to clearly distinguish distinct areas of overlap for a set of views, by using a separate connector view for each area of overlap.

- It generalizes elegantly for cases where more than two views share an overlap, and for cases where we want to indicate overlaps between the connector views themselves.

- It provides a more flexible platform for incorporating various types of preferences into the merge process [7]. A simple example are layout preferences: we may want to choose a certain layout for the parts that are in common between a set of views and preserve that layout in the merge. Layout preferences require explicit models for the shared parts.
- Connectors can be used as requirements baselines when the scope of the elicitation process widens. For example, if the elicitation process starts with two stakeholders and a new stakeholder emerges later on, it is natural for the third stakeholder to use the connectors between the views of the first two stakeholders as baselines for further elaboration of his/her own views.

8.4 Beyond Equality Relationships

To have a flexible view exploration process, we may need to express certain kinds of non-equality relationships between view elements. A notable example of such relationships is *similarity* stating that two or more elements are similar in some respects but not equivalent. Expressing non-equality relationships can be made possible by extending the formalism of interest with new modeling constructs. ERDs, for example, have been extended with a special notation for denoting similarities between schema elements [7]. Incorporating such constructs into merge scenarios is straight-forward. For example, in Figure 13, we could elect to introduce a similarity node instead of the Meeting request be sent goal node to state that Send request by email and Send request by snail mail are conceptually similar without providing details about the nature of the similarity.

9 Related Work

Over the years, the term "view" (or "viewpoint") has appeared in the literature with several different meanings. Views have been used to mean different classes of users [33], the contexts in which different roles are performed [34], to distinguish between stakeholder terminologies [35], and to encapsulate knowledge about a system into loosely-coupled objects [36]. A survey and comparison of the existing view-based approaches can be found in [37]. Our interpretation of views falls closely in line with the emerging trends in model management where views are employed to capture conceptual data gathered from disparate sources into *independent* but *interrelated* units. To describe view interrelationships, explicit mappings must be defined between them [3].

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 [2, 9, 10]. 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 views as loosely-coupled distributed objects [2].

The use of multi-valued logics for merging incomplete and inconsistent views was first proposed in [38] and was later generalized in [17]. The latter work, which serves as a precursor to this paper, suggested category theory as a unified means for formalizing incompleteness and inconsistency and characterizing the merge operation. However, the work was conceived primarily as part of a framework for supporting automated reasoning over state-machines. This paper instead focuses on using the same algebraic principles for devising a model management framework for requirements elicitation.

Schema merging [4, 7, 8] is an important operation in database design for combining disparate schemata, and has been identified as one of the core activities in metadata management [3, 39]. Our framework extends the syntactic aspects of the state-of-the-art on schema merging in several respects: Firstly, the existing schema merging approaches only support the three-way merge pattern whereas our framework can handle arbitrary interconnection patterns. Secondly, our framework can explicitly model inconsistencies and allows for the deferral of their resolution. This is in contrast to the above-cited work where inconsistencies are not tolerated and need to be resolved as soon as discovered. Thirdly, our framework is parameterized by a meta-model and can hence be applied to various modeling formalisms; but, schema merging is, to a great extent, tailored to entity relationship diagrams or similar schema modeling notations.

In [40], a combination of natural language processing and formal concept analysis techniques has been employed for formalizing and visualizing requirements extracted from multiple use case descriptions. Reconciliation is done through defining a thesaurus for unifying the vocabularies of the extracted concepts. The work also attempts to strategize handling of inconsistencies; but, it does not support explicit modeling of stakeholders' beliefs; and further, falls short of accounting for requirements evolution and describing its impacts on the management of inconsistencies.

The closest work to ours in the area of Requirements Engineering is the merging framework of [5]. In this work, views are described by graph transformation systems and colimits are used to merge them. However, the work cannot handle inconsistent views.

Recently, a framework has been proposed for merging partial behavioral models [41]. There, stakeholders' models are described by partial state transition systems and are merged based on the process-algebraic refinement relations between them. The work supports incompleteness and can also detect inconsistencies; however, the merge operation fails when the models are inconsistent. Another difference between our approach and the work is that they do not explicitly describe view correspondences and rely on bi-similarity relations to give the relationships between the states of different views. This can make it difficult for requirements analysts to guide the merge process as they cannot directly hypothesize the merge alternatives.

Annotated graphs bear similarity to fuzzy graphs [42]. The work on fuzzy graphs is focused on the analysis of isolated models using graph-theoretic techniques, whereas in our work, the focus is on describing the structural relationships between models using algebraic techniques.

The ability to trace requirements back to their human sources is cited as one of the most important traceability concerns in software development [43]. To this end, *contribution structures* [44] have been proposed as a way to facilitate cooperative work among teams and to ensure that the contributions of involved parties are properly accounted for throughout the entire development life-cycle. The notions of origin and stakeholder traceability in our work try to address a similar problem in the context of view merging.

The importance of establishing traceability links between artifacts and the assumptions involved in creating them has been emphasized in design rationale [45, 46] and design traceability [47]. However, the focus of that work has been mainly on assumptions that relate upstream and downstream artifacts. Our work, instead, focuses on requirements elicitation which is an entirely upstream activity. We discuss the nature of the relationships between views produced during elicitation, and provide an approach for keeping track of how each assumption made about view interrelations affects the merge.

10 Conclusions and Future Work

We have presented 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 to trace contributions in the merged view back to their sources and to the relevant merge assumptions. 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 is a major part of our ongoing work and is a step toward applying the work to large-scale conceptual modeling. We are also investigating possible ways for adding a semantics-aware component to the framework for sanity-checking the merges. Another interesting venue is studying whether our framework can be used for relating the *behaviors* of models. The interconnections used in our approach are based on homomorphisms and the fact that homomorphisms have been employed in various abstraction frameworks [48] 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 thing that can be studied. We also plan to develop a more usable version of the tool to investigate how well it supports cooperative conceptual modeling, and especially stakeholder negotiation during requirements analysis.

Acknowledgments. We thank John Mylopoulos, Renée Miller, Pamela Zave, Sotirios Liaskos, Yijun Yu, Linda Liu, Faye Baron, and Shiva Nejati for helpful discussions. We thank the members of the Formal Methods, Database, and EarlyRE groups at the University of Toronto, as well as the anonymous reviewers of the RE'05 Conference and the RE Journal for their insightful comments. Financial support was provided by NSERC, MITACS, and BUL.

References

[1] S. Easterbrook, E. Yu, J. Aranda, Y. Fan, J. Horkoff, M. Leica, and R. Qadir. Do viewpoints lead to better conceptual models? an exploratory case study. In *Proceedings of the 13th International Requirements Engineering Conference*, pages 199–208, 2005.

[2] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.

[3] P. Bernstein. Applying model management to classical meta data problems. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research*, pages 209–220, 2003.

[4] P. Buneman, S. Davidson, and A. Kosky. Theoretical aspects of schema merging. In *Proceedings of the*

3rd International Conference on Extending Database Technology, pages 152–167, 1992.

[5] H. Ehrig, G. Engels, R. Heckel, and G. Taentzer. A combined reference model- and view-based approach to system specification. *International Journal of Software Engineering and Knowledge Engineering*, 7(4):457–477, 1997.

[6] S. Castano and V. De Antonellis. Deriving global conceptual views from multiple information sources. In *Conceptual Modeling – Current Issues and Future Directions*, volume 1565 of *Lecture Notes in Computer Science*, pages 44–55. Springer, 1999.

[7] R. Pottinger and P. Bernstein. Merging models based on given correspondences. In *Proceedings of 29th International Conference on Very Large Data Bases*, pages 862–873., 2003.

[8] S. Melnik, E. Rahm, and P. Bernstein. Rondo: a programming platform for generic model management. In *SIGMOD Conference*, pages 193–204, 2003.

[9] S. Easterbrook and B. Nuseibeh. Using viewpoints for inconsistency management. *Software Engineering Journal*, 11(1):31–43, 1996.

[10] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer. Flexible consistency checking. *ACM Transactions on Software Engineering and Methodology*, 12(1):28–63, 2003.

[11] E. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *Proceedings of the 3rd International Symposium on Requirements Engineering*, pages 226–235, 1997.

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

[13] M. Sabetzadeh and S. Easterbrook. An algebraic framework for merging incomplete and inconsistent views. In *Proceedings of the 13th International Requirements Engineering Conference*, pages 306–315, 2005.

[14] M. Sabetzadeh and S. Easterbrook. iVuBlender: A tool for merging incomplete and inconsistent views. In *Proceedings of the 13th International Requirements Engineering Conference*, pages 453–454, 2005. Tool Demo Paper.

[15] M. Sabetzadeh and S. Easterbrook. Traceability in viewpoint merging: A model management perspective. In *Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering*, November 2005. To appear.

[16] A. van Lamsweerde, R. Darimont, and P. Massonet. The meeting scheduler system – problem statement. ftp://ftp.info.ucl.ac.be/pub/publi/92.

[17] M. Sabetzadeh and S. Easterbrook. Analysis of inconsistency in graph-based viewpoints: A category-theoretic approach. In *Proceedings of the 18th International Conference on Automated Software Engineering*, pages 12–21, 2003.

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

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

[20] H. Ehrig and G. Taentzer. Computing by graph transformation, a survey and annotated bibliography. *Bulletin of the European Association for Theoretical Computer Science*, 59:182–226, 1996.

[21] G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: Foundations*, volume 1. World Scientific, River Edge, NJ, USA, 1997.

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

[23] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3–4):241–265, 1996.

[24] The goal-oriented requirement language (GRL). http://www.cs.toronto.edu/km/GRL.

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

[26] N. Belnap. A useful four-valued logic. In G. Epstein and J. Dunn, editors, *Modern Uses of Multiple-Valued Logic*, pages 5–37. Reidel, Dordrecht, Netherlands, 1977.

[27] M. Ginsberg. Bilattices and modal operators. In *Proceedings of the 3rd Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 273–287, 1990.

[28] C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

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

[30] J. Madhavan, P. Bernstein, and E. Rahm. Generic schema matching with cupid. In *Proceedings of 27th International Conference on Very Large Data Bases*, pages 49–58, 2001.

[31] S. Melnik, H. Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings of the 18th International Conference on Data Engineering*, pages 117–128, 2002.

[32] A. van Lamsweerde, R. Darimont, and P. Massonet. Goal-directed elaboration of requirements for a meeting scheduler: Problems and lessons learnt. In *Proceedings of the 2nd International Symposium on Requirements Engineering*, pages 194–203, 1995.

[33] D. Ross. Applications and extensions of SADT. *IEEE Computer*, 18(4):25–34, 1985.

[34] S. Easterbrook. Domain modeling with hierarchies of alternative viewpoints. In *Proceedings of the 1st International Symposium on Requirements Engineering*, pages 65–72, 1993.

[35] R. Stamper. Social norms in requirements analysis: an outline of MEASUR. In M. Jirotka and J. Goguen, editors, *Requirements engineering: social and technical issues*, pages 107–139. Academic Press, London, UK, 1994.

[36] A. Finkelsetin, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–58, 1992.

[37] P. Darke and G. Shanks. Stakeholder viewpoints in requirements definition: A framework for understanding viewpoint development approaches. *Requirements Engineering*, 1(2):88–105, 1996.

[38] S. Easterbrook and M. Chechik. A framework for multi-valued reasoning over inconsistent viewpoints. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 411–420, 2001.

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

[40] D. Richards. Merging individual conceptual models of requirements. *Requirements Engineering*, 8(4):195–205, 2003.

[41] S. Uchitel and M. Chechik. Merging partial behavioural models. In *Proceedings of the 12th International Symposium on Foundations of Software Engineering*, pages 43–52, 2004.

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

[43] O. Gotel and A. Finkelstein. Extended requirements traceability: results of an industrial case study. In *Proceedings of the 3rd International Symposium on Requirements Engineering*, pages 169–178, 1997.

[44] O. Gotel and A. Finkelstein. Contribution structures (requirements artifacts). In *Proceedings of the 2nd International Symposium on Requirements Engineering*, pages 100–107, 1995.

[45] G. Fischer, A. Lemke, R. McCall, and A. Morch. Making argumentation serve design. In T. Moran and J. Carroll, editors, *Design rationale: concepts, techniques, and use*, pages 267–293. Lawrence Erlbaum Associates, Mahwah, NJ, USA, 1996.

[46] T. Gruber and D. Russell. Generative design rationale: beyond the record and replay paradigm. In T. Moran and J. Carroll, editors, *Design rationale: concepts, techniques, and use*, pages 323–349. Lawrence Erlbaum Associates, Mahwah, NJ, USA, 1996.

[47] A. Egyed. A scenario-driven approach to traceability. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 123–132, 2001.

[48] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 19(2):1512–1542, 1994.