Merging Models Based on Given Correspondences

University of Washington Technical Report UW-CSE-03-02-03

Rachel A. Pottinger
University of Washington
Seattle, WA 98195-2350 USA
rap@cs.washington.edu

Philip A. Bernstein
Microsoft Research
Redmond, WA 98052-6399 USA
philbe@microsoft.com

Abstract

A model is a formal description of a complex application artifact, such as a database schema, an application interface, a UML model, an ontology, or a message format. The problem of merging such models lies at the core of many meta data applications, such as view integration, mediated schema creation for data integration, and ontology merging. This paper examines the problem of merging two models given correspondences between them. It presents requirements for conducting a merge and a specific algorithm that subsumes previous work.

1 Introduction

A *model* is a formal description of a complex application artifact, such as a database schema, an application interface, a UML model, an ontology, or a message format. The problem of merging models lies at the core of many meta data applications, such as view integration, mediated schema creation for data integration, and ontology merging. In each case, two given models need to be combined into one. Because there are many different types of models and applications, this problem has been tackled independently in specific domains many times. Our goal is to provide a generic framework that can be used to merge models in all these contexts.

Combining two models requires first determining correspondences between the two models and then merging the models based on those correspondences. Finding correspondences is called schema matching; it is a major topic of ongoing research and is not covered here [12, 14, 16, 19]. Rather, we focus on the problem of combining the models after correspondences are established. We encapsulate the problem in an operator, Merge, which takes as input two models, A and B, and a mapping Map_{AB} between them that embodies the given correspondences, and returns a third model that is the "duplicate-free union" of A and B with respect to Map_{AB}. This is not as simple as set union because the models have structure, so the semantics of "duplicates" and duplicate removal may be complex. In addition, the result of the union can manifest constraint violations, called *conflicts*, that Merge must repair.

An example of the problems addressed by Merge can be seen in Figure 1. It shows two representations of Actor, each of which could be a class, concept, table, etc. A mapping between A and B is shown by the dashed lines. In this case, it seems clear that Merge is meant to collapse A.Actor and B.Actor into a single element, and similarly for Bio. Clearly, A.ActID should be merged with B.ActorID, but what should the resulting element be called? And what about the actor's name? Should the merged model represent the actor's name as one element (ActorName), two elements (FirstName and LastName), three elements (ActorName with FirstName and LastName as children), or in some other way?

These cases of differing representations between input models are called conflicts. For the most part, conflict resolution is independent of how A and B are represented. Yet most work on merging schemas concentrates on doing it in a data-model-specific way, revisiting the same problems for ER variations [30], XML [4], data warehouses [11], semi-structured data [5], or relational and object-oriented databases [10]. Note that these works, like ours, consider merging only the models, not the instances of the models. Some models, such as ontologies and ER diagrams, have no instance data, and merging the models is a necessary precursor to merging those models with instance data.

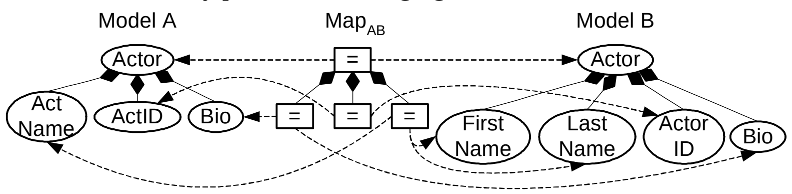


Figure 1: Examples of models to be merged

The similarities among these solutions offer an opportunity for abstraction. One important step in this direction was an algorithm for schema merging and conflict resolution of models by Buneman, Davidson, and Kosky (hereafter *BDK*) [10]. Given a set of pair-wise correspondences between two models that have Is-a and Has-a relationships, BDK give a formal definition of merge and show how to resolve a certain kind of conflict to produce a unique result. We use their theoretical algorithm as a base, and expand the range of correspondences, model representations, conflict types, and applications, yielding a robust and practical solution.

Merge is one of the operators proposed in [6] as part of *model management*, a framework that consists of operators for manipulating models and mappings. This paper is a proposed semantics and implementation of that operator. Other model management operators include: Match, which returns a mapping between two given models; Apply, which applies a given function to all the elements of a model; and Diff, which given two models, returns a model consisting of all items in the first model that are not in the second model [6].

The main contribution of this paper is the design of a practical generic merge operator. It includes the following specific contributions:

- Generic requirements for merge that every design should satisfy.
- The use of an input mapping that is a first-class object, enabling us to express richer correspondences than previous approaches.
- A characterization of when Merge can be automatic.
- A taxonomy of the conflicts that can occur and a definition of conflict resolution strategies using the mapping's richer correspondences.
- Experiments that show our approach scales to a large real world application.
- An analysis that shows our approach subsumes previous merge work.

The paper is structured as follows: Section 2 gives a precise definition of Merge. Section 3 describes our categorization of conflicts that arise from combining two models. Section 4 describes how to resolve conflicts in Merge, often automatically. Section 5 defines our merge algorithm. Section 6 discusses the associativity and commutativity of Merge. Section 7 discusses an alternate merge definition and how to simulate it using Merge and other model management operators. Section 8 evaluates Merge experimentally by merging two large anatomy databases and conceptually by showing how our approach subsumes previous work. Section 9 is the conclusion.

2 Problem Definition

2.1 Representation of Models

Defining a representation for models requires (at least) three meta-levels. Using conventional meta data terminology, we can have: a *model*, such as the database schema for a billing application; a *meta-model*, which consists of the type definitions for the objects of models, such as a meta-model that says a relational database schema consists of table definitions, column definitions, etc.; and a *meta-meta-model*, which is the representation language in which models and meta-models are expressed, usually an extended entity-relationship model. We do not use the term "data model," because it is potentially ambiguous. For example, in a relational database system, the relational data model is the meta-meta-model, while in a model management system, it is one of many meta-models (which may include meta-models for ODMG schemas and UML models).

The goal of our merge operator, Merge, is to merge two models based on a mapping between them. For now, we discuss Merge using a small meta-meta-model (which we extend in Section 4.1). It consists of the following:

- *Elements* with semi-structured properties. Three properties are required: Name, ID, and History. Name and ID are self-explanatory. History describes the last operator that acted on an element, which allows tracing the lineage of operators applied to the element.
- Binary, directed, typed *relationships* with cardinality constraints. A relationship is a connection between two elements. We enumerate relationship types in Section 4.1. Relationships can be either explicitly present in the model or *implied* according to the rules governing models. For example, the meta-meta-model rules may state that "a is a b" and "b is a c" implies that "a is a c." Relationship cardinalities are omitted from the figures for ease of exposition.

In Figure 1 elements are shown as nodes, the value of the Name property is the node's label, mapping relationships are edges with arrowheads, and sub-element relationships are diamond-headed edges.

2.2 Merge Inputs

The inputs to Merge are the following:

- Two models: A and B.
- A mapping, Map_{AB}, which is a model that defines how A and B are related.

- An optional designation that one of A or B is the *preferred model*. When Merge faces a choice that is not specified in the mapping, it chooses the option from the preferred model, if there is one.
- Optional overrides for default Merge behavior.

We have found it to be quite important to allow the input mapping to be a first-class model, so it can consist of elements and relationships. To explain why, we first define the structure of mappings.

A mapping is a model that additionally allows us to relate the elements of two other models through *mapping elements*. A mapping element is like any other element except that it also can be the origin of a *mapping relationship*, M(x, y), which specifies that the origin element, x, *represents* the destination element y. So a given mapping element, x, represents all elements y such that M(x, y). A different mapping between the models in Figure 1 is shown in Figure 2

Mapping elements come in two types: equality and similarity. Each mapping element has a property HowRelated, with value "Equality" or "Similarity," to distinguish the two types. An *equality mapping element* x asserts that for all y1, y2 ∈ Y such that M(x, y1) and M(x, y2), y1 = y2. All elements that are represented by the same equality mapping element are said to *correspond* to one another. A *similarity mapping element* x asserts that the set of all y1, y2 ∈ Y such that M(x, y1) and M(x, y2) are related through a complex expression. The expression can be encoded in the Expression property of x, which is a property of all similarity mapping elements.

Given this rich structure for mappings, complex relationships can be defined between elements in A and B, not just simple correspondences. For example, the mapping in Figure 2 (which is between the same models as Figure 1) shows that the FirstName and LastName of model B should be sub-elements of the ActorName element of model A; this is expressed by element m_4, which corresponds to the ActorName in A and contains the elements m_5 and m_6 which correspond to FirstName and LastName in B. Prior algorithms, which operate without a first class mapping, cannot express this kind of relationship. Often, they require user intervention during Merge to incorporate relationships that are more complicated than simply equating two elements.

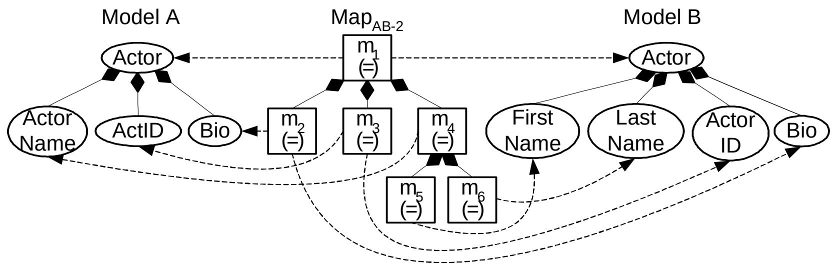


Figure 2: Mapping specifying that FirstName and LastName should be sub-elements of ActorName

Although not shown in Figure 2, a mapping can express *similarity* between elements in A and B. For example, if Bio in model A is a French translation of Bio in model B and this needs to be reflected explicitly in the merged model, they might be connected by a similarity mapping element whose Expression property explains how the two Bio values are related.

A mapping can also contain non-mapping elements that do not correspond directly to elements in either A or B but are useful in describing how elements in A and B are related. For example, if models A and B described French and American actors respectively, the mapping could denote that both Actor elements are specializations of an element, AllActors, that appears in neither A nor B.

2.3 Merge Semantics

The output of Merge is a model that retains all non-duplicated information in A, B, and Map_{AB}; it collapses information that is declared redundant by Map_{AB}. If we consider the mapping to be a third model, this definition corresponds to the least-upper-bound defined in BDK [10], "a schema that presents all the information of the schemas being merged, but no additional information." We also require Merge to be generic in the sense that it does not require its inputs or outputs to adhere to any given meta-model. We consider other merge definitions in Section 7.

We now define the semantics of Merge more precisely. The function "Merge(A, Map_{AB}, B) → G" merges two models A and B based on a mapping Map_{AB}, which describes how A and B are related, producing a new model G that satisfies the following Generic Merge Requirements (GMRs):

1. **Element preservation:** Each element in the input has a corresponding element in G. Formally: each element $e \in A \cup B \cup \text{Map}_{AB}$ corresponds to exactly one element $e' \in G$. We define this correspondence as $\chi(e, e')$.
2. **Equality preservation:** Input elements are mapped to the same element in G if and only if they are equal in the mapping, where equality in the mapping is transitive. Formally: two elements $s, t \in A \cup B$ are said to be *equal* in Map_{AB} if there is an element $v \in A \cup B$ and an equality mapping element x such that M(x, s) and M(x, v), where either

v = t or v is equal to t in Map_{AB}. If two elements s, t ∈ A ∪ B are equal in Map_{AB}, then there exists a unique element e ∈ G s.t. χ(s, e) and χ(t, e). If s and t are not equal in Map_{AB}, then they correspond to different elements in G.

3. **Relationship preservation:** Each input relationship is explicitly in or implied by G. Formally: for each relationship r(s, t) ∈ A ∪ B ∪ Map_{AB} where s, t ∈ A ∪ B ∪ Map_{AB} and r is not a mapping relationship M(s, t) with s ∈ Map_{AB} and t ∈ A ∪ B, if χ(s, s′) and χ(t, t′), then either s′ = t′, r(s′, t′) ∈ G, or r(s′, t′) is implied in G.
4. **Similarity preservation:** Elements that are declared to be similar (but not equal) to one another in Map_{AB} retain their separate identity in G and are related to each other by some relationship. More formally, for each pair of elements s, t ∈ A ∪ B, where s and t are connected to a similarity mapping element, x, in Map_{AB} and s and t are not connected to an equality mapping element, there exist elements e, s′, t′ ∈ G such that χ(s, s′), χ(t, t′), r(e, s′), r(e, t′), χ(x, e), and e describes an expression relating s and t.
5. **Meta-meta-model constraint satisfaction:** G satisfies all constraints of the meta-meta-model. G may include elements and relationships in addition to those specified above that help it satisfy these constraints. Note that we do not require G to conform to any meta-model.
6. **Extraneous item prohibition:** Other than the elements and relationships specified above, no additional elements or relationships exist in G.
7. **Property preservation:** For each element e ∈ G, e has property p if and only if ∃ t ∈ A ∪ B ∪ Map_{AB} s.t. χ(t, e) and t has property p.
8. **Value preference:** The value, v, of a property p, for an element e is denoted e(p) = v. The value of e(p) is chosen from mapping elements corresponding to e if possible, else from the preferred model if possible, else from any element that corresponds to e. More formally:
   - T = {t | χ(t, e)}
   - J ={j ∈ (T ∩ MapAB) | j(p) is defined}
   - K ={k ∈ (T ∩ the preferred model) | k(p) is defined}
   - M ={m ∈ T | m(p) is defined}
     - If J ≠ ∅, then e(p) = j(p) for some j ∈ J
     - Else if K ≠ ∅, then e(p) = k(p) for some k ∈ K
     - Else e(p) = m(p) for some m ∈ M

GMR 8 illustrates our overall conflict resolution strategy: give preference first to the option in the mapping (i.e., the explicit user input into the merged outcome), then to the preferred model, and then choose arbitrarily. The ID, History, and HowRelated properties are determined differently as discussed in Section 5

For example, the result of merging the models in Figure 2 is shown in Figure 3. Note that the relationships Actor-FirstName and Actor-LastName in model B are implied by transitivity in Figure 3, so GMR 3 is satisfied.

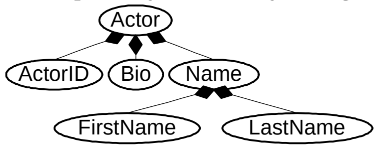


Figure 3: The result of performing the merge in Figure 2

The GMRs are not always satisfiable. For example, if there are constraints on the cardinality of relationships that are adjacent to an element, then there may be no way to preserve all relationships. Depending on the relationships and meta-meta-model constraints, there may be an automatic resolution, manual resolution or no possible resolution that adheres to the GMRs. In Section 4 we discuss conflict resolution for a set of common constraints and when such resolution can be automatic. We also specify default resolution strategies for each type of constraint and note when resolution can be made to adhere to the Generic Merge Requirements outlined above.

3 Conflict Resolution

Determining the merged model requires resolving conflicts in the input. We categorize conflicts based on the meta-level at which they occur:

- **Representation conflicts** (Section 3.1) are caused by conflicting representations of the same real world concept – a conflict at the *model level*. Resolving these conflicts requires manual user intervention. Such conflict resolution is necessary for many uses of mappings – not just Merge. Hence we isolate it from Merge by requiring it to be captured in the input mapping.
- **Meta-model conflicts** (Section 3.2) are caused by the constraints in the *meta-model* (e.g., SQL DDL). Because adhering to a specific meta-model is not required by a generic Merge, we resolve them with another operator after Merge is done.

- **Fundamental conflicts** (Section 3.3) are caused by constraints in the meta-meta-model. These conflicts must be resolved to ensure that the merge result is a model that conforms to the meta-meta-model. Unlike representation conflicts, they only occur after the models are merged, so the mapping cannot be required to solve them; hence they must be resolved in Merge.

3.1 Representation Conflicts

A representation conflict arises when two models describe the same concept in different ways. For example, in Figure 1 model A represents Name by one element, ActorName, while model B represents it by two elements, FirstName and LastName. After merging the two models, should name be represented by one, two or three elements? The decision is application dependent.

In Merge we resolve representation conflicts using the input mapping. Having a mapping that is a model allows us to specify that elements in models A and B are either:

- The same, by connecting them to the same equality mapping element. Thus Merge can collapse the elements into one element that includes all relationships incident to the elements in the conflicting representations.
- Related by relationships and elements in our meta-meta-model. E.g., we can model FirstName and LastName in A as sub-elements of ActorName in B by the mapping shown in Figure 2.
- Related in some more complex fashion that we cannot represent using the relationship types in our meta-meta-model. E.g., we can represent that ActorName equals the concatenation of FirstName and LastName by a similarity mapping element that has mapping relationships incident to all three and an Expression property describing the concatenation. Resolution can be done by a later operator that understands the semantics of Expression.

The mapping can also specify the values of properties. For example, if model B were French, it might name the concept of actor "acteur." The mapping can say whether the merged element should be named "actor," "acteur," or something else entirely, or it can leave it unspecified.

Solving representation conflicts has been a focus of the ontology merging literature [22, 23] and of database schema matching [12, 16, 19].

3.2 Meta-model Conflicts

A meta-model conflict occurs when the merge result violates a meta-model-specific (e.g., SQL DDL) constraint. For example, suppose that in Figure 2 Actor is a SQL table in model A, an XML database in model B, and a SQL table in the merged model. If the mapping in Figure 2 is used, there will be a meta-model conflict in the merge result because SQL DDL has no concept of sub-column. This does not violate any principle about the *generic* merged outcome.

If a meta-model constraint violation arises in a merge result, that result must be modified to correct the violation. Traditionally, merge results are required to conform to a given meta-model during the merge. However, since Merge is meta-model independent, we do not resolve this type of conflict in Merge. Instead, we break coercion out as a separate step, so that Merge remains generic and the coercion step can be used independently of Merge. We therefore introduce an operator, EnforceContraints, that coerces a model to obey a set of constraints. This operator is necessarily meta-model specific. However, it may be possible to implement it in a generic way, driven by a declarative specification of each meta-model's constraints. We leave this as future work.

3.3 Fundamental Conflicts

The third and final type of conflict is called a fundamental conflict. It occurs above the meta-model level at the meta-meta-model level, the representation that all models must adhere to. A fundamental conflict occurs when the result of Merge would not be a model due to violations of the meta-meta-model. This is unacceptable because later operators would be unable to manipulate it.

One possible meta-meta-model constraint is that an element has at most one type. We call this the *one type restriction*. Given this constraint, an element with two types manifests a fundamental conflict. For example in the model fragments in Figure 4(a) ZipCode has two types: Integer and String. In the merge result in Figure 4(b), the two ZipCode elements are collapsed into one element. But the type elements remain separate, so ZipCode is the origin of two type relationships.

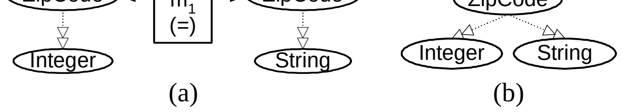


Figure 4: A merge that violates the one type restriction

Since Merge must return a well-formed instance of the meta-meta-model, it must resolve fundamental conflicts. Resolution rules for some fundamental conflicts have been proposed, such as [10] for the one type restriction. We have identified other kinds of fundamental conflicts and resolution rules for them which we describe in Section 4 and incorporate into our generic Merge.

Alternate merge semantics might ignore GMR 1 (Element preservation) and resolve fundamental conflicts by requiring the mapping to include preferences to resolve the conflict. For example, the mapping in Figure 4 could specify that the type of ZipCode is String, and Merge should ignore the conflicting information that ZipCode is of type Integer. In general we do not recommend this strategy because it loses information (e.g., that ZipCode is of type Integer). However, since Merge allows users to specify alternate resolutions, as discussed in Section 4, this strategy can be easily incorporated by specifying it as the resolution strategy for one type conflicts.

The choice of meta-meta-model therefore is integrally related to Merge. However, since we are skeptical that there is a meta-meta-model capable of solving all meta data management problems, we chose the following approach: We define the properties of Merge in a meta-meta-model independent way. We then define fundamental conflict resolution for a meta-meta-model that includes many of the popular semantic modeling constructs. Finally we describe other typical meta-meta-model conflicts and provide conflict resolution strategies for them.

4 Resolving Fundamental Conflicts

The meta-meta-models we consider are refinements of the one described in Section 2.1. Section 4.1 describes *Vanilla*, an extended entity-relationship-style meta-meta-model that includes many of the popular semantic modeling constructs. Section 4.2 describes our merging strategy, both for Vanilla and for relationship constraints that may be used in other meta-meta-models.

4.1 The Vanilla Meta-Meta-Model

Elements are first class objects with semi-structured properties. Name, ID, and History are the only required properties. Properties are used to describe information that is relevant to meta data operations, such as Match and Merge.

Note that these are properties of the element viewed as an instance, not as a template for instances. For example, suppose an element e represents a class definition, such as Person. Viewing e as an instance, its Name property has the value “Person.” To enable instances of Person to have a property called Name (thereby viewing e as a template for an instance), we create a relationship from e to another element whose Name property is “Name.”

Relationships are binary, directed, typed, and ordered (as in XML), and have an optional cardinality constraint. A relationship type is one of "Associates", "Contains", "Has-a", "Is-a", and "Type-of" (described below). Reflexive relationships are disallowed in Vanilla. We assume that between any two elements there is at most one relationship of a given type and cardinality pairing.

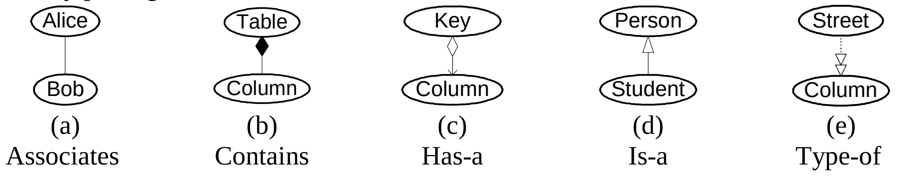


Figure 5: Different relationship types in Vanilla

There are cases where the previous restriction is inconvenient. For example, one might want two kinds of Has-a relationships between "Movie" and "Person", namely "director" and "actor". This can be handled either by specializing Person into two sub-elements, or by reifying the director and actor Has-a relationships (i.e., turn the relationships into objects), which is the choice used in Vanilla. We disallow multiple named Has-a relationships of the same cardinality between two elements because it leads to a need for correspondences between named relationships of different models. E.g., if the director and actor relationships are called "réalisatuer" and "acteur" in another model, we need a relationship between director and réalisatuer and between actor and acteur. These correspondences would complicate the meta-meta-model. The same expressiveness is gained by reifying relationships, thereby avoiding this complexity.

A relationship R(x, y) between elements x and y may be a mapping relationship, M(x, y), described earlier, or one of the following:

- Associates - A(x, y) means x is Associated with y. This is the weakest relationship that can be expressed. It has no constraints or special semantics. Figure 5(a) says that Alice is Associated with Bob.
- Contains - C(x, y) means *container* x Contains *containee* y. Intuitively, a containee cannot exist on its own; it is a part of its container element. Operationally, this means that if all of the containers of an element, y, are deleted, then y must be deleted. Contains is a transitive relationship and must be acyclic. If C(x, y) and x is in a model M, then y is in M as well. Figure 5(b) says that Table Contains Column.

- Has-a - H(x, y) means x Has-a sub-component y (sometimes called "weak aggregation"). Has-a is weaker than Contains in that it does not propagate delete and can be cyclic. Figure 5(c) says that Key Has-a Column.
- Is-a - I(x, y) means x Is-a specialization of y. Like Contains, Is-a is transitive, acyclic, and implies model membership. Figure 5(d) says that Student Is-a Person.
- Type-of - T(x, y) means x is of type y. Each element can be the origin of at most one Type-of relationship (the one type restriction described in Section 3.3). Figure 5(e) says that the Type-of Street is Column.

Vanilla has the following *cross-relationship-type implications* that imply relationships based on explicit ones:

- If T(q, r) and I(r, s) then T(q, s)
- If I(p, q) and H(q, r) then H(p, r)
- If I(p, q) and C(q, r) then C(p, r)
- If C(p, q) and I(q, r) then C(p, r)
- If H(p, q) and I(q, r) then H(p, r)

In Vanilla, a model L is a triple (E_L, $Root_L$, R_L) where E_L is the set of elements in L, $Root_L \in E_L$ is the root of L, and R_L is the set of relationships in L. Given a set of elements E and relationships R (which may include mapping relationships), membership in L is determined by applying the following rules to $Root_L \in E$, adding existing model elements and relationships until a fixpoint is reached (i.e., until applying each rule results in no new relationships):

- I(x, y), $x \in E_L \rightarrow y \in E_L$; if a specialization x is in the model, then its generalization y is in the model
- C(x, y), $x \in E_L \rightarrow y \in E_L$; if a container x is in the model, then its containee y is in the model
- T(x, y), $x \in E_L \rightarrow y \in E_L$; if an element x is in the model, then its type y is in the model
- R(x, y), $x \in E_L$, $y \in E_L \rightarrow$ R(x, y) $\in R_L$
- M(x, y), $x \in E_L \rightarrow$ M(x, y) $\in R_L$

Since a mapping is a model its elements must be connected by relationships indicating model containment (Contains, Is-a, or Type-of). However, since these relationships obfuscate the mapping, we often omit them from figures when they do not affect Merge's behavior.

In what follows, when we say relationships are "implied", we mean "implied by transitivity and cross-relationship-type implication."

We define two models to be *equivalent* if they are identical after all implied relationships are added to each of them until a fixpoint is reached (i.e., applying each rule results in no new relationships). For example, the two models in Figure 6 are equivalent:

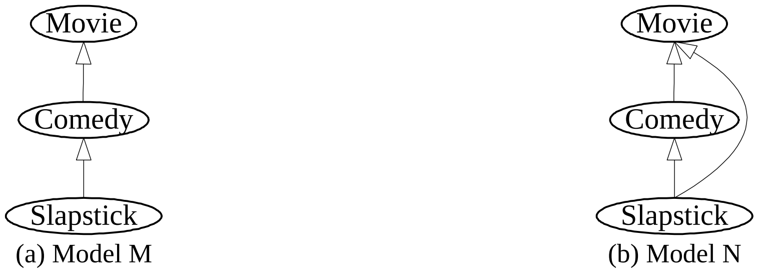


Figure 6: Model M is a minimal covering of model N

A *minimal covering* of a model is an equivalent model that has no edge that is implied by the union of the others. For example, the model in Figure 6(a) is a minimal covering of the model in Figure 6(b). A model can have more than one minimal covering.

To ensure that the merge result G is a model, we require that $Root_{MapAB}$ is an equality mapping element with M($Root_{MapAB}$, $Root_A$) and M($Root_{MapAB}$, $Root_B$), and that $Root_{MapAB}$ is the origin of no other mapping relationships.

4.2 Meta-Meta-Model Relationship Characteristics and Conflict Resolution

This section explores fundamental conflicts in Merge with respect to both Vanilla and other meta-meta-models: what features lead to an automatic Merge, when manual intervention is required, and default resolutions. Since the default resolution may be inadequate due to application specific requirements, Merge allows the user to either (1) specify an alternative function to apply to each conflict resolution type or (2) resolve the conflict manually.

Vanilla has only two fundamental constraints (i.e., that can lead to fundamental conflicts): (1) the one type restriction, and (2) the Is-a and Contains relationships must be acyclic. These fundamental conflicts can be resolved fully automatically in Vanilla.

4.2.1 Relationship-Element Cardinality Constraints

Many meta-meta-models restrict some types of relationships to a maximum or minimum number of occurrences incident to a given element. For example, the one type restriction says that no element can be the origin of more than one Type-of relationship. Such restrictions can specify minima and/or maxima on origins or destinations of a relationship of a given type.

Cardinality Constraints in Vanilla - Merge resolves one type conflicts using a customization of the BDK algorithm [10] for Vanilla; a discussion of the change can be found in Appendix B. Recall Figure 4 where the merged ZipCode element has both Integer and String types. The BDK resolution creates a new type that inherits from both Integer and String and replaces the two Type-of relationships from ZipCode by one Type-of relationship to the new type, as shown in Figure 7. Note that both of the original relationships (ZipCode is of type Integer and String) are implied.

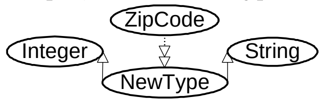


Figure 7: Resolving the one type conflict of Fig. 4

This solution creates a new element, NewType in Figure 7, whose Name, ID, and History properties must be determined. The ID is assigned an unused ID value, and the Name is set to be the names of the elements it inherits from, delineated by a slash; so NewType in Figure 7 is named "Integer/String." The History property records why the element came into existence, in this case, that Merge created it from the elements Integer and String.

This approach to resolution of one type conflicts is an example of a more general approach: to resolve a conflict, alter explicit relationships so that they are still implied and the GMRs are still satisfied. Thus, the more implication rules in the meta-meta-model, the easier conflict resolution is.

In Vanilla, a model G *satisfies model membership* if all elements of G are reachable from G's root by following containment relationships: Is-a, Contains, and Type-of. Hence requiring that G, the output of Merge, is a model is a form of a minimum element-relationship cardinality. We know that, excluding conflict resolution, G is a model because:

1. $\chi(\text{Root}_A, \text{Root}_G)$, $\chi(\text{Root}_B, \text{Root}_G)$, $\chi(\text{Root}_{MapAB}, \text{Root}_G)$ from the input and GMR 2 (Equality preservation).
2. Root_G is not the destination of any relationships (and hence is a candidate to be root) because of GMR 6 (Extraneous item prohibition) and because it only corresponds to Root_A, Root_B, and Root_{MapAB} which likewise are roots.
3. each element g in G can be determined to be a member of the model with root Root_G because model membership is determined through following the containment semantics of the meta-meta-model. Each element e such that $\chi(e, g)$ must be a member of A, B, or Map_{AB}. Assume without loss of generality that $e \in A$. Then there must be a path P of elements and relationships from Root_A to e that determines that e is in A. By GMR 1 (Element preservation) and GMR 3 (Relationship preservation), a corresponding path P′ must exist in G, and hence g is a member of the model with root Root_G.

Hence, conflict resolution notwithstanding, G is guaranteed to satisfy model membership without Merge using any special conflict resolution. After conflict resolution for Vanilla, G still satisfies model membership; the BDK solution to the one type restriction only adds relationships and elements that adhere to model containment. The acyclic resolution only collapses a cycle, which cannot disturb the model membership of the remaining element.

Cardinality Constraints in General - There are two kinds of relationship-element cardinality constraints: for some *n*: (1) at least *n* relationships of a given type must exist (*minimality constraints*) and (2) at most *n* relationships of a given type may exist (*maximality constraints*).

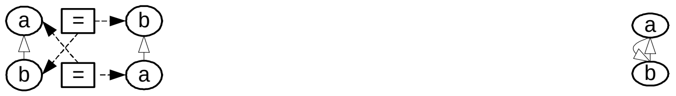
Since Merge (excluding conflict resolution) preserves all relationships specified in the input, the merged model is guaranteed to adhere to the minimality constraint, too. Merge guarantees that any minimality constraint imposed on the model will be imposed on the merged result. For example, one potential minimality constraint is that each element must be the origin of one Type-of relationship. If this were the case, then each of the input models, A, B, and Map_{AB} would have to obey the constraint. Hence each element in A, B, and Map_{AB} would be the origin of at least one Type-of relationship. Since Merge preserves the relationships incident on each element, each element in G is also the origin of at least one Type-of relationship. Conflict resolution may break this property, so conflict resolution strategies must consider these kinds of constraints.

More care is required for maximality constraints, such as the one type restriction. If they occur in a meta-meta-model, the generic merge attempts resolution by removing redundant relationships. Next, the default Merge resolution would look for a cross-type implication rule that can resolve the conflict (i.e., apply the default resolution strategy). If no such rule exists, then we know of no way to resolve the conflict while still adhering to the GMRs. To continue using the one type restriction as an example, first we calculate a minimal covering of the merged model and see if it still has a one type restriction conflict. If so, then we apply a cross-type implication rule (if T(q, r) and I(r, s) then T(q, s)) which allows us to resolve the conflict and still adhere to the GMRs.

4.2.2 Cyclicity

Many meta-meta-models require some relationship types to be acyclic. In Vanilla, Is-a and Contains must be acyclic. In this section, we first consider acyclic constraints in Vanilla and then acyclicity constraints in general. We do not separately consider non-reflexive constraints, which are similar but less restrictive; the techniques for acyclic relationships can be applied to them as well.

Acyclicity in Vanilla - Merging the example in Figure 8 (a) would result in Figure 8 (b) which has a cycle between elements a and b. Since Is-a is transitive, a cycle of Is-a relationships implies that all of the elements in the cycle are equal. Thus, Merge's default solution is to collapse the cycle into a single element. As with all conflicts, users can override with a function or manual resolution. To satisfy GMR 7 (Property preservation), the resulting merged element contains the union of all properties from the combined elements. GMR 8 (Value preference) dictates the value of the merged element's properties.



(a) (b)

Figure 8: Merging the models in (a) causes the cycle in (b)

Acyclicity Constraints in General - If the constrained relationship type is not transitive, collapsing the cycle would not retain the desired semantics in general (although it does work for cycles of length two). The resolution is to see if any cross-relationship-type implications allow all relationships to exist implicitly without violating the acyclicity constraint. If so, the conflict can be resolved automatically. Without such a relationship implication it is impossible to merge the two models while retaining all of the relationships; either some default resolution strategy must be applied that does not retain all relationships, or human intervention is required.

4.2.3 Other Relationship Conflicts

The following are conflict types that may occur in meta-meta-models other than Vanilla:

- Meta-meta-models with multiple instance levels may impose restrictions on the merged model; for example, a specialization hierarchy may not cross meta-levels.
- If a meta-meta-model allows only one relationship of a given type between a pair of elements, the cardinality of the relationship must be resolved if there is a conflict. For example, in Figure 9 what should be the cardinality of the Contains relationship between Actor and ActID? 1:n? m:1? m:n? One could argue that it should be m:n because this is the most general, however this may not be the desired semantics. Since any resolution of this conflict is going to result in the loss of information and therefore will not adhere to GMR 3 (Relationship preservation), no generic resolution of this type can adhere to the GMRs.

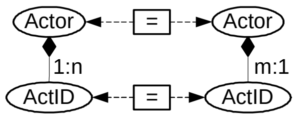


Figure 9: Merging multiple cardinalities

- If only one set of specializations of an element may be declared disjoint, merging two orthogonal such sets requires conflict resolution, e.g., if actors are specialized as living/dead in one model and male/female in another.

5 The Merge Algorithm

This section describes an algorithm for Merge that satisfies the GMRs. Steps 1-4 calculate the duplicate-free union of models A and B and mapping Map_{AB}, to form a merged model G. The use of Map_{AB} resolves representation conflicts. Meta-meta-model constraints are enforced in step 5. Merge works as follows:

1. **Initialize** the merge result G to ∅.
2. **Elements:** Group the elements of A, B, and Map_{AB}. Initially each element is in its own group. Then:
   a. If a relationship M(d, e) exists between an element $e \in (A \cup B)$ and a mapping equality element $d \in Map_{AB}$, then combine the groups containing d and e.
   b. After iterating (a) to a fixpoint, create a new element in G for each group.
3. **Element Properties:** Let e be a merged element in G corresponding to a group I. The value v of property p of e, e(p), is defined as follows:
   a. Excluding the property HowRelated, the properties of e are the union of the properties of the elements of I. Merge determines the values of properties of e other than History, ID, and HowRelated as follows:

      $J = \{j \in (I \cap Map_{AB}) \mid j(p) \text{ is defined}\}$

K ={k ∈ (I ∩ the preferred model) | k(p) is defined}
M ={m ∈ I | m(p) is defined}

i. If J ≠ ∅, then e(p) = j(p) for some j ∈ J
ii. Else if K ≠ ∅, then e(p) = k(p) for some k ∈ K
iii. Else e(p) = m(p) for some m ∈ M

It is guaranteed that some value for each property of e will exist from the definition of the properties of e. In (i) – (iii) if more than one value is possible, then one is chosen arbitrarily.

b. Property e(ID) is set to an unused ID value. Property e(History) describes the last action on e. It contains the operator used (in this case, Merge) and the ID of each element in I. This implicitly connects the Merge result to the input models and mapping without requiring an explicit relationship between them.

c. An element e in G is a mapping element if and only if some element in I is in (A ∪ B) and is a mapping element (i.e., A and/or B is a mapping). Hence in this case, e(HowRelated) is defined; this is the only exception to GMR 7 (Property preservation). Its value is determined by GMR 8 (Value preference).

4. **Relationships:**

For every two elements e and f in distinct groups P and R of G, where P and R do not contain similarity elements, if there exists p ∈ P and r ∈ R such that R(p, r) is of type t and has cardinality c, then create a (single) relationship R(e, f) of type t and cardinality c. Reflexive mapping relationships (i.e., mapping relationships between elements that have been collapsed) are excluded since they no longer serve a purpose. For example, without this exception, after the Merge in Figure 2 is performed, the mapping relationship between elements ActorName and m_4 would be represented by a reflexive mapping relationship with both relationship ends on ActorName. However, this relationship is redundant, so we eliminate it from G.

a. If element e corresponds to a similarity mapping element in Map_{AB}, replace each mapping relationship, M, whose origin is e by a Has-a relationship whose origin is e's group and whose destination is M's destination's group. For example, if the two Bio elements in Figure 2 were connected by a similarity mapping element instead of an equality element, the result would be as in Figure 10.

b. Relationships originating from an element are ordered as follows:

- Those that correspond to relationships in Map_{AB}.
- Those that correspond to relationships in the preferred model but not in $Map_{AB.}$
- All other relationships.

Within each category, relationships appear in the order they appear in the input.

c. Finally, Merge removes implied relationships from G until a minimal covering remains.

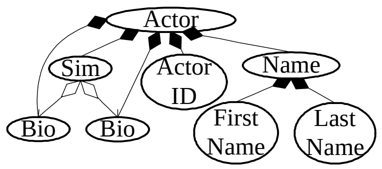


Figure 10: Results of the Merge in Figure 2 if the Bio elements were connected by a similarity mapping element

5. **Fundamental conflict resolution:** After steps (1) – (4) above, G is a duplicate-free union of A, B, and Map_{AB}, but it may not satisfy fundamental constraints, the meta-meta-model conflicts that must be resolved during Merge. For each fundamental conflict, if a special resolution strategy has been defined then apply it. If not, apply the default resolution strategy as described in Section 4.2.

Resolving one conflict may interfere with another, or even create another. This does not occur in Vanilla; while resolving the one type restriction does create Is-a relationships, they cannot be cyclic since the origin is new and thus cannot be the destination of another Is-a relationship. However, if conflict resolution interference is a concern in some other meta-meta-model, then Merge can apply create a priority scheme by assuming that the conflict resolutions are an ordered list. The conflict resolutions are then applied until fixpoint. Since resolving one type conflicts cannot create cycles in Vanilla, conflict resolution in Vanilla is guaranteed to terminate. However, conflict resolution rules in other meta-meta-models must be examined to avoid infinite loops.

The algorithm described above adheres to the GMRs in Section 2.3. We can see this as follows:

- Step 1 (Initialization) initializes G to the empty set.
- Step 2 (Elements) enforces GMR 1 (Element preservation). It also enforces the first direction of GMR 2 (Equality preservation); elements equated by Map_{AB} are equated in G. No other work is performed in step 2.

- Step 3 (Element properties) performs exactly the work in GMR 7 (Property preservation) and GMR 8 (Value preference) with the exceptions of the refinements in steps 3b and 3c for the ID, History, and HowRelated properties. No other work is performed in step 3.
- In step 4 (Relationships), step 4a enforces GMR 3 (Relationship preservation) and step 4b enforces that a relationship exists between elements mapped as similar, as required in GMR 4 (Similarity preservation). Step 4d removes only relationships that are considered redundant by the meta-meta-model. Step 4c (relationship ordering) is the only step not explicitly covered by the GMR, and it does not interfere with any other GMRs.
- Step 5 (Fundamental conflict resolution) enforces GMR 5 (Meta-meta-model constraint satisfaction) and performs no other work.

If special resolution strategies in step 5 do no work to violate any GMR or equate any elements not already equated, GMRs 2 (Equality preservation), 4 (Similarity preservation) and 6 (Extraneous item prohibition) are satisfied, and all GMRs are satisfied. Other than special properties (ID, History, and HowRelated) and the ordering of relationships, no additional work is performed beyond what is needed to satisfy the GMRs.

6 Properties of Multiple Merges

Since meta data operations seldom occur in isolation, the properties of sequences of merges must be examined, namely associativity and commutativity. Section 6.1 examines the commutativity of Merge. Section 6.2 examines the associativity of Merge. Section 6.3 discusses commutativity and associativity when the order of merges affects the choice of mappings that are used to drive each merge. For ease of exposition we only consider cases where the outcome is uniquely specified by the inputs (e.g., exactly one correct choice of value exists for each property).

6.1 Commutativity

We say that Merge is commutative if, for any pair of models M and N and any mapping Map_{MN} between them, Merge(M, Map_{MN}, N) = Merge(N, Map_{MN}, M) = G. We assume that (1) the same model is the preferred model in each Merge and (2) if there are unspecified choices to be made (e.g., choosing a property value from among several possibilities, each of which is allowed by Merge) the same choice is made in both Merges. We begin by showing that commutativity holds for Merge as specified by the GMRs and then show that it holds for the Merge algorithm specified in Section 5.

The commutativity of Merge as specified by the GMRs in Section 2.3 follows directly from their definition, since they are symmetric: Rules 1-4 and 6-7 are inherently symmetric. Rule 8 (Value preference) is symmetric as long as the preferred model is the same in both Merges and unspecified choices are the same in both Merges, as stipulated in (2) above. Rule 5 is the resolution of fundamental conflicts. In Vanilla this is symmetric since collapsing all cycles and resolving one type violations using the BDK algorithm are both symmetric[1]. Hence in Vanilla the GMRs are symmetric and thus commutative. However, Merge in other meta-meta-models may not be commutative, depending on their conflicts resolution rules.

The algorithm specification in Section 5 is commutative as well; again we show this from the algorithm's symmetry. Steps 1 (Initialize) and 2 (Elements) are symmetric. Steps 3 (Element properties) and 4 (Relationships) are symmetric as long as the preferred model is the same in both merges and arbitrary choices are the same, as stipulated in (2) above. Step 5 (Fundamental conflict resolution) is symmetric if the conflict resolutions are symmetric. As argued above, this holds for conflict resolution in Vanilla, and hence the Merge algorithm is symmetric and thus commutative in Vanilla.

6.2 Associativity

We say that two models are *isomorphic* if there is a 1:1 onto correspondence between their elements, and they have the same relationships and properties (but the values of their properties and the ordering of their relationships may differ). Merge is associative if, for any three models M, N, and O, and any two mappings Map_{MN} (between M and N) and Map_{NO} (between N and O), R is isomorphic to S where:

P = Merge(M, Map_{MN}, N)
Q = Merge(N, Map_{NO}, O)
Map_{PN} = Match(P, N)
Map_{NQ} = Match(N, Q)
R = Merge(P, Compose(Map_{PN}, Map_{NO}), O)
S = Merge(M, Compose(Map_{MN}, Map_{NQ}), Q)

[1] Appendix B gives the details of our modifications to the BDK algorithm.

Match(P, N) and Match(N, Q) are meant to compute χ as defined in the GMRs by matching the IDs of N with the IDs in the History property of P and Q respectively. N, P, Q, Match(P, N), and Match(N, Q) are shown in Figure 11.

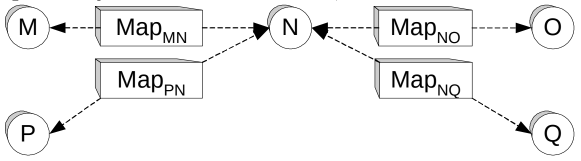


Figure 11: To show associativity we must create intermediate mappings using the Match operator

The Compose operator takes a mapping between models A and B and a mapping between models B and C and returns the composed mapping between A and C. Consider Compose(Map_{PN}, Map_{NO}). Intuitively it must transfer each mapping relationship of Map_{NO} having a destination in N to a relationship having a destination in P. Since Map_{PN} maps each element in N to exactly one element in P, any Compose operator will provide this functionality (such as the one described in Appendix A). Compose(Map_{MN}, Map_{NQ}) operates similarly.

A *morphism* is a set of directed *morphism relationships* from elements of one graph to elements of another. To show that the two final merged models R and S are isomorphic, we define a morphism $\varphi(R \rightarrow S)$ and show that (i) φ is 1:1 and onto, (ii) $R(x, y) \in R_R$ if and only if $R(\varphi(x), \varphi(y)) \in R_S$, and (iii) x has property p if and only if $\varphi(x)$ has property p. We initially consider the result of Merge ignoring the fundamental conflict resolution. We phrase the argument in terms of the GMRs. We do not repeat the argument for the algorithm, for the following reason: The end of Section 5 shows that the algorithm maintains all of the GMRs and that the only additional work done by the merge algorithm beyond the GMRs is (1) to order the relationships and (2) set the values of the HowRelated and ID properties. The latter two additions do not affect the isomorphism, so we do not repeat the associativity argument for the algorithm.

We create φ as follows. First we create the morphisms shown as arrows in Figure 12 by using Match and Compose the same way they were used to create R and S. We refer to the morphisms in Figure 12 that start at R as $Morph_R$ and the morphisms that end at S as $Morph_S$. Next we create five morphisms from R to S by composing $Morph_R$ with $Morph_S$. φ is the duplicate-free union of the five morphisms from the previous step.

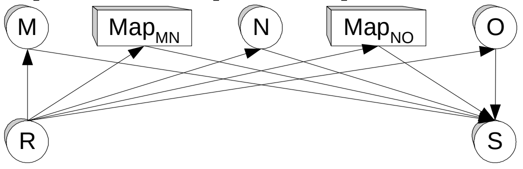


Figure 12: Initial morphisms created to show associativity

We want to show that φ is an isomorphism from R to S. We first show that φ is onto (i.e., for all $y \in S$, there exists $x \in R$ such that $\varphi(x) = y$):

1. Let T be the set of elements in M, Map_{MN}, N, Map_{NO}, and O.
2. From GMRs 1 (Element preservation) and 2 (Equality preservation) and the definitions of Match and Compose, we know that each element in T is the destination of exactly one morphism relationship in $Morph_R$. I.e., each element in M, Map_{MN}, N, Map_{NO}, and O corresponds to exactly one element in the merged model. From GMR 6 (Extraneous item prohibition) and the definitions of Match and Compose we know that each element in R is the origin of at least one morphism relationship to T. I.e., each element in R must correspond to some element in M, Map_{MN}, N, Map_{NO}, or O. Recall that we are not considering conflict resolution, so there will be no elements introduced due to GMR 5 (Meta-meta-model constraint satisfaction).
3. Similarly each element of T is the origin of exactly one morphism relationship in $Morph_S$ and each element in S is the destination of at least one $Morph_S$ morphism relationship from T.
4. Hence from steps 2 and 3 and the definitions of Match and Compose, φ is onto.

Next we show that φ is 1:1(i.e., for all $x_1, x_2 \in R$, $\varphi(x_1) = \varphi(x_2) \rightarrow x_1 = x_2$).

1. From the definition of φ, the only way for φ to not be 1:1 is if:
   a. Some element $r \in R$ is the origin of more than one morphism relationship in $Morph_R$ or
   b. Some element $s \in S$ is the origin of more than one morphism relationship in $Morph_S$.
2. If statement 1a is true, then from GMR 2 (Equality preservation) and the definitions of Match and Compose, r must be the result of merging some elements from T that were equal in some mapping. Similarly, if statement 1b is true, then from GMR 2 (Equality preservation) and the definitions of Match and Compose, s must be the result of merging some elements from T that were equal in some mapping. We now must show that the equating of the elements is associative and hence for element r in step 1a, each morphism relationship in φ that begins with r will end at the same element in S, thus providing a duplicate morphism relationship and not one that contradicts that φ is 1:1.

3. The equating of elements is associative; this follows directly from the grouping strategy in Merge step 2 (Element properties).
   a. If elements are not equated by a mapping, then they will never be merged into the same object.
   b. Hence the only interesting case is when elements from three different models are mapped to one another; take the example of elements $r \in M$, $t \in N$, $v \in O$ as shown in Figure 13. Given the equality mapping elements m_1 and m_3, r, t, and v are merged into one element, regardless of which order the models are combined. If, however, both relationships implied similarity, then all three elements will exist. If one relationship implied similarity (say r similar to t) and the other equality (say t equals v), then the resulting elements are the same regardless of order; elements representing r and t are combined, and two elements representing t and v still exist separately.
4. Hence the equating of the elements is associative and φ is 1:1.

Next we must show that $R(x, y) \in R_R$ if and only if $R(\varphi(x), \varphi(y)) \in R_S$. That is, a relationship exists in R if and only if a corresponding relationship exists in S. GMR 3 (Relationship preservation) guarantees that each relationship R input to Merge has a corresponding relationship R′ in the merged model unless R's origin and destinations have been collapsed into a single element. Similarly the Match and Compose definitions preserve the elements, and a relationship $R(x, y) \in R_R$ if and only if $R(\varphi(x), \varphi(y)) \in R_S$

The last step to show that φ is an isomorphism is to show that each element $r \in R$ has property p if and only if φ(r) has property p. GMR 7 (Property preservation) implies that each element in the merged model has a property p if and only if some input element that it corresponds to has property p. From the argument showing that φ is 1:1, we know that the equating of elements is associative, and hence $r \in R$ has property p if and only if φ(r) has property p. Hence φ is an isomorphism from R to S and Merge is associative.

Note that Merge is not associative with respect to the *values* of properties. Their value is determined as explained in GMR 8 (Value preference). After a sequence of Merges, the final value of a property may depend on the order in which the Merges are executed. This occurs because the value assigned by the last Merge in the sequence can overwrite the values of any Merges that preceded it. For example, in Figure 13 the mapping element m_1 in Map_{MN} specifies the value a for property Bio. In addition, the Merge definition specifies that O is the preferred model for the merge of N and O. If the sequence of operators is:

Merge(M, Map_{MN}, N) → P
Merge(P, Compose(Match(P, N),Map_{NO}), O) → R

Then in model P the Bio property as a result of merging r and t will have the Bio value a since it is specified in m_1. In the second Merge, model O will be the preferred model, and the value of the Bio property of the resulting element will be c.

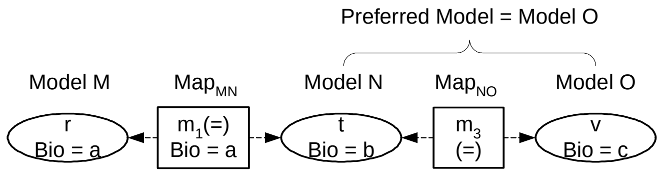


Figure 13: A series of mappings and models

However, if the sequence of operators is:

Merge(N, Map_{NO}, O) → Q
Merge(M, Compose(Map_{MN}, Match(N, Q)), Q) → S

Then the Bio property of the element that corresponds to t and v will have the value c since model O is the preferred model. Since the value of Bio in mapping element m_1 is a, the final result will have a as the value of Bio instead of c as in the first example.

Unless Merge can express a total preference between models – which is impractical – it will not be associative with respect to the final values of properties.

Hence, ignoring conflict resolution, Merge is associative. Since all of the fundamental conflict resolution in Vanilla is associative, Merge is associative for Vanilla as well (see [10] for references on the associativity of the BDK).

6.3 Mapping-independent Commutativity and Associativity

We say that Merge is *mapping-independent commutative* (resp. *associative*) if it is commutative (resp. associative) even when the order of Merge operations affects the choice of mapping that is used in each Merge. For example, consider the models and mappings in Figure 14. In (a), Map_{MN} is the only mapping that equates elements s and u. When Map_{MN} is used, as in (b), elements s and u are combined. However, when Map_{MN} is not used, as in (c), s and u remain as separate elements.

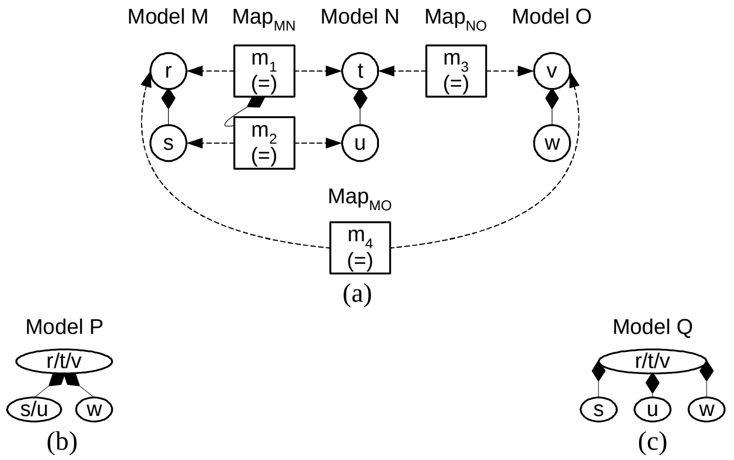


Figure 14: (a) A set of models and mappings. (b) the result of merging the models using Map$_{MN}$ and Map$_{NO}$. (c) the results of merging the models using Map$_{NO}$ and Map$_{MO}$.

When is Merge guaranteed to be mapping-independent associative and commutative? Ignoring meta-meta-model constraint satisfaction, given a set of models, S (e.g., {M, N, O} in Figure 14), and two sets of mappings Mappings$_A$ (e.g., {Map$_{MN}$, Map$_{NO}$}) and Mappings$_B$ (e.g., {Map$_{NO}$, Map$_{MO}$}) over S, in order for Merge to produce isomorphic results it must be the case that:

- Elements r and v are equated to one another either directly or transitively in Mappings$_A$ if and only if they are equated to one another directly or transitively in Mappings$_B$; r can be declared equal to t and t equal to v in one set of mappings and in another set of mappings r can be declared equal to v and v equal to t.
- Elements r and v are declared to be "similar to" another element in Mappings$_A$ if and only if they are declared to be "similar to" the same element in Mappings$_B$.
- Additional elements and relationships are introduced in Mappings$_A$ if and only if corresponding elements and relationships are introduced in Mappings$_B$.

Informally, we know that these are the only requirements because:

- Merge is associative and commutative if the mappings are the same, as shown above.
- Mappings have three roles with respect to Merge; they can (1) declare elements to be equal (2) declare elements to be similar or (3) add in additional elements and relationships. We address each of these three roles below:
- **Equality:** Because equality is transitive, we only need to enforce that elements that are equated in one set of mappings are also equated in the other.
- **Similarity:** Mappings$_A$ and Mappings$_B$ must both declare that the same elements are similar if they are to be isomorphic to each other. However, since similarity is not transitive, if similarity is used then there is an implicit restriction on the sets of mappings; if Mappings$_A$ declares an element in model S$_1$ to be similar to an element in model S$_2$, then Mappings$_B$ must contain a mapping between S$_1$ and S$_2$ in order for the similarity relationship to be expressed. We do not need to consider the more complicated case when one mapping declares two elements to be similar through a similarity mapping element, s, and then another similarity mapping element, t, declares s to be similar to some other element because by our problem definition the set of mappings cannot map results of previous Merges.
- **Additional elements and relationships:** Finally, because mappings can also add elements and relationships, if Mappings$_A$ adds an element or a relationship, then Mappings$_B$ must add a corresponding element or relationship as well. However, as with similarity, there may be an implicit restriction on the set of mappings; if Mappings$_A$ declares an element in model S$_1$ to contain an element in model S$_2$, then Mappings$_B$ must contain a mapping between S$_1$ and S$_2$ in order for the Contains relationship to be expressed. Again, because the set of mappings cannot map results of previous merges, we need not consider more complicated cases.

7 Alternate Merge Definitions

Many alternate merge definitions can be implemented using our Merge operator in combination with other model management operators. In this section we consider a few useful variations. In each case the goal is to take models A and B, and a mapping, Map$_{AB}$ between them, and merge them to produce a model G.

7.1 Merge Only Elements Specifically Mentioned in the Mapping

Some applications want to merge only elements specifically mentioned in the mapping. One such example is Subject Oriented Programming [25], which uses Merge to combine classes. After classes are combined, some variables should not be merged because they are private. This formulation of Merge can be implemented by:

DeepCopy(Map_{AB}) → $Map_{AB'}$, A′, B′
Apply(A′, a function to delete elements not mapped by $Map_{AB'}$)
Apply(B′, a function to delete elements not mapped by $Map_{AB'}$)
Merge(A′, $Map_{AB'}$, B′) → G

where DeepCopy is a variant of the Copy operator that copies the mapping as well as the models that it connects [7].

7.2 Three-Way Merge

Three-way merge, a common merging problem that occurs in file versioning and computer supported collaborative work [2]. Given a model and two different modified versions of it, the goal is to merge the modified versions into one model.

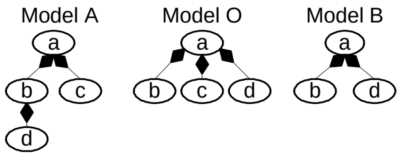


Figure 15: A three-way merge assuming name equality. Model O is the common ancestor of models A and B.

For example, consider Figure 15 where model O has been modified in two different ways to create both models A and B. Suppose there are mappings between O and A and between O and B based on element name equivalence. Notice that in A, element d has been moved to be a child of element b, while in B the element c has been deleted.

There are several variations of three-way merge which arise due to different treatments of an element that was modified in one model and deleted or modified in the other. All of these are expressible using model management operators. One variation assumes that elements deleted in one model but modified in the other should be included in the merged model. More precisely it assumes that the merged model L should have the following properties:

- If an element e was added in A or B, then e is in L.
- If an element e is present and unmodified in A, B, and O, then e is in L.
- If an element e was deleted in A or B and unmodified or deleted in the other, then e is not be in L.
- If an element e was deleted in A or B and modified in the other, then e is in L (because by modifying e the model designer has shown that e is still of interest).
- If an element e was modified in A or B and unmodified in the other, then the modified version of e is in L.
- If an element e was been modified in A and B, then conflict resolution is needed to determine what is in L.

This 3-way merge can be implemented as follows. We determine equality for elements in A and B based on the History property.

1. Create a mapping Map_{AB} between A and B such that:
   a. If a ∈ A and b ∈ B are equal, a mapping element expressing equality between a and b is added to Map_{AB}.
   b. If an element e exists in each of O, A, and B, and a property of e has been changed in exactly one of A or B, then Map_{AB} says to use the changed property value.
2. Create model D such that if an element or relationship has been deleted in one of A or B and is unmodified in the other, it is included in D.
3. G = Merge(A, Map_{AB}, B).
4. Return Diff(G, D).

Note that this does not handle a new element x created independently in A and B. To allow this, a new mapping could be created to relate A.x and B.x.

Creating the information contained in Map_{AB} and D can be done using a sequence of model management operators. Appendix C shows this in detail.

Most algorithms for three-way merge have (1) a “preferred” model that breaks ties and (2) a method for resolving conflicts such as when an element is deleted in one descendent model and modified in the other. We support the former with Merge’s preferred model the latter by applying the model management Apply operator.

8 Evaluation

Our evaluation has two main goals: Section 8.1 shows that Merge can be applied to a real world application where it scales to large models and discovers relevant conflicts and Section 8.2 shows that our Merge definition subsumes previous work.

8.1 Applying Merge to Large Ontologies

We tested Merge on a large bioinformatics application to show that Merge scales to large models and uncovers real conflicts caused by merging such large models. The problem is to merge two models of human anatomy: the Foundational Model of Anatomy (FMA) [27], which is designed to model anatomy in great detail, and the GALEN Common Reference Model [26], which is designed to aid clinical applications. These are very large models. As expressed in a variant of Vanilla, FMA contains 895,307 elements and 2,032,020 relationships, and GALEN contains 155,307 elements and 569,384 relationships; both of the models were larger in the Vanilla variant than in their "native" format since many of their relationships required reification. The two models have significant structural differences (e.g., some concepts expressed in FMA by three elements are expressed in GALEN by four elements), so merging the two is challenging. Note that there is no additional instance information for either model.

A database researcher familiar with FMA, GALEN, and model management took 13 weeks to import the models into a variant of Vanilla and create a mapping consisting of 6265 correspondences. The mapping is small relative to the model sizes since the models have different goals and thus different contents. It contains only 1-to-1 correspondences, so we were unable to test our hypothesis that having the mapping as a first class model enables more accurate merging. Hence we concentrated on three other issues: (1) few changes to Vanilla and Merge would be needed to merge the models, even though Merge was not tailored for this domain, (2) Merge would function on models this large, and (3) the merged result would not be simply read from the mapping (i.e., the conflicts that we anticipated would occur).

For the first issue, the researcher needed to add to Vanilla two relationship types: Contains-t(x, y), which says that x *can contain* instances of y, and Has-t(x, y), which says that x *can have* instances of y. In effect, these are meta-model definitions where x and y are types. Neither relationship type led to new kinds of fundamental conflicts. Also, the one type restriction was not relevant to the anatomists. Hence, the only change to Merge's default behavior was to list the two new types and ignore the one type restriction.

Merging these models took approximately 20 hours on a Pentium III 866 with 1 GB of RAM. This is an acceptable amount of time since Merge would only be run occasionally in a relatively long project (13 weeks in our case). The merge result before fundamental conflict resolution had 1,045,411 elements and 2,590,969 relationships. 9,096 relationships were duplicates, and 1,339 had origins and destinations that had been equated.

Since the input mapping only uses 1-to-1 correspondences we would expect most elements in the merged model to correspond to exactly two elements in FMA and GALEN. However, 2344 merged elements correspond to exactly three elements in FMA and GALEN, and 623 correspond to more than 3 elements. One merged element corresponds to 1215 elements of GALEN and FMA.

The anatomists verified that the specialization hierarchy should be acyclic, as it was in both inputs. However, before conflict resolution the merge result contained 338 cycles in the specialization hierarchy, most of length 2. One was of length 18.

The anatomists agreed that the result of the merge was useful both as a final result, assuming that the input mapping was perfect, and as a tool for determining possible flaws in the input mapping. Exploring the former is a largely manual process and is the subject of ongoing medical informatics research.

8.2 Comparison to Previous Approaches

There has been considerable work on merge in other contexts and applications. An important result of our work is that it subsumes previous literature on merge. In this section we show how Merge, assisted by other model management operators, can implement previous approaches to generic merging (Section 8.2.1), view integration (Section 8.2.2), and ontology merging (Section 8.2.3) even though it is not tailored to their meta-models.

8.2.1 Generic Merging Algorithms

BDK provides the basis for our work: their algorithm creates the duplicate free union of two models based on name equality of the models' elements. Their meta-meta-model contains elements with a name property and two relationship types, Is-A and Has-a, where Has-a must obey the one type restriction.

Essentially Merge encompasses all of the BDK work by taking the duplicate free union of two models and then applying the one type restriction resolution. Their work considers neither any other meta-meta-model conflicts, nor resolutions when their solution to the one type conflict is inappropriate. In addition, BDK cannot resolve representation conflicts because it lacks the explicit mapping that allows it to do so. Further details of how Merge corresponds to the BDK algorithm can be found in Appendix B.

Rondo [20] is a model management system prototype that includes an alternate Merge definition based entirely on equality mappings. Two elements can be declared to be equal, and each 1-1 mapping relationship can specify a preference for one element over another. Like our Merge and BDK's, Rondo essentially creates the duplicate-free union of the elements and relationships involved. Some conflicts require removing elements or relationships from the merged model (e.g., if a SQL column belongs to two tables in a merge result, it must be deleted from one of them). Just as our Merge resolves such meta-model conflicts later, Rondo does such resolutions in a separate operator.

Our Merge is richer than Rondo's in several respects:

1. Our Merge allows more precise resolution of representation conflicts; the structure in the mapping allows elements in the input to be related in some fashion other than equivalence.
2. Our Merge allows specification and resolution of fundamental conflicts in the Merge operator itself rather than pushing the work to a separate manual operator.
3. By specifying that a choice is first taken from the mapping, then the preferred model, then any model, our Merge allows for some preferences to be made once per Merge rather than at each mapping relationship.

8.2.2 View Integration

View integration is the problem of combining multiple user views into a unified schema [3]. This problem has been studied in many contexts [4, 5, 8, 11, 18, 28, 29]. View integration algorithms (1) ensure the merged model contains all of the objects in the two original models, (2) reconcile representation conflicts in the views (e.g., if a table in one view is matched with a column in another), and (3) require user input to guide the merge.

Spaccapietra and Parent have a well known algorithm [30] that consists of a set of rules and a prescribed order to apply them in. Their meta-meta-model, ERC+, has three different object types: attributes, entities, and relations. An entity is an object that is of interest on its own. An attribute describes data that is only of interest while the object it characterizes exists. A relation describes how objects in the model interact. ERC+ has three kinds of relationships: Is-a, Has-a, and May-be-a, which means that an object may be of that type.

Vanilla can encode ERC+ by representing attributes, entities and relations as elements. ERC+ Is-a relationships are encoded as Vanilla Is-a relationships. ERC+ Has-a relationships are encoded as Vanilla Contains relationships (the semantics are the same). To encode in Vanilla the May-be-a relationships originating at an element e, we create a new type t such that Type-of(e, t) and for all f such that e May-be-a f, Is-a(t, f).

The Spaccapietra and Parent algorithm for merging models can be implemented using model management by encoding their conflict resolution rules either directly into Merge or in mappings.

Below, we summarize each of their rules and how it is covered by GMRs to merge two ERC+ diagrams A and B to create a new diagram, G. Again we use $\chi(e, e')$ to say that $e \in A \cup B$ corresponds to an element $e' \in G$.

1. Objects integration – If $a \in A$, $b \in B$, $a = b$, and both a and b are not attributes, then add one object g to G such that $\chi(a, g)$ and $\chi(b, g)$. Also, if a and b are of differing types, then g should be an entity. This corresponds to GMR 1 (Element preservation) plus an application of the EnforceConstraints operator to coerce the type of objects of uncertain type into entities.
2. Links integration – If there exist relationships R(p, c) and R(p′, c′), where $p, c \in A$, $p', c' \in B$, $p = p'$, $c = c'$, $\chi(p, g)$, $\chi(p', g)$, $\chi(c, t)$, and $\chi(c', t)$ (i.e., two parent-child pairs are mapped to one another), where neither g nor t are attributes, R(g, t) is added to G. This is covered by GMR 3 (Relationship preservation).
3. Paths integration rule – Exclude implied relationships from the merged model. This is covered by GMR 3 (Relationship preservation) and Merge step 4d (Relationships: removing implied relationships) in the algorithm.
4. Integration of attributes of corresponding objects – If there exist relationships R(p, c) and R(p′, c′) where $p, c \in A$, p', $c' \in B$, $p = p'$, $c = c'$, $\chi(p, g)$, $\chi(p', g)$ (i.e., two parent-child pairs are mapped to one another), and c and c′ are attributes, then add an attribute t to G such that $\chi(c, t)$, $\chi(c', t)$ and R(g, t). This is covered by GMRs 2 (Equality preservation) and 3 (Relationship preservation).
5. Attributes with path integration – if for some attributes $c \in A$ and $c' \in B$, $c = c'$, there is no relationship R such that R(p, c) and R(p′, c′) where $p = p'$ (i.e., c and c′ have different parents), add an element g to G such that $\chi(c, g)$, $\chi(c', g)$, and add all relationships necessary to attach g to the merged model. If one of the relationship paths is implied and the other is not, add only the non-implied path. This is covered by GMR 1 (Element preservation) and 3 (Relationship preservation).
6. Add objects and links without correspondent – All objects and relationships that do not correspond to anything else are added without a correspondent. This is covered by GMR 1 (Element preservation) and 3 (Relationship preservation).

8.2.3 Ontology Merging

The merging of ontologies is another model merging scenario. A frame-based *ontology* specifies a domain-specific vocabulary of objects and a set of relationships among them. These objects may have properties and relationships with other objects. The two relationships are Has-a and Is-a. Ontologies include constraints (called *facets*), but they were ignored by all algorithms that we studied. We describe here PROMPT [22], originally called SMART [23], which combines ontology matching and merging.

PROMPT focuses on driving the match, since once the match has been found, their merge is straightforward. As in Merge, their merging and matching begin by including all objects and relationships in either model. As the match proceeds, objects that are matched to one another are collapsed into a single object. Then PROMPT suggests that objects, properties, and relationships that are related to the merged objects may match (e.g., if two objects each with a "color" property have been merged, it suggests matching those "color" properties).

Like many merge algorithms, PROMPT includes the notion of a preferred model. It also has two modes, override and merge. In override mode PROMPT chooses the option from the preferred model, while in merge mode the user is prompted for input.

PROMPT keeps track of its state with two lists: *Conflicts* and *ToDo*. *Conflicts* lists the current model's conflicts with the meta-model. *ToDo* keeps track of suggested matches. A more complete description of the PROMPT algorithm is shown in Appendix D.

Our algorithm allows us to provide as much merging support as PROMPT. In the merge of two models, A and B, to create a new model G, PROMPT has the following merge functionality, which we relate to our GMRs. We do not consider PROMPT's match functionality since it is outside Merge's scope.

1. Each set of objects $O \in A \cup B$ whose objects $o \in O$ have all been matched to each other correspond to one object in G. This is covered by GMR 2 (Equality preservation).
2. Each object $o \in A \cup B$ that has not been matched to some other object corresponds to its own object in G. This is covered by GMR 2 (Equality preservation).
3. An object $g \in G$ consists of all of the properties of the objects in A or B that correspond to it. This is covered by GMR 7 (Property preservation).
4. If a conflict exists on some property's name or value, it is resolved either (1) by the user which corresponds to the user input in Merge's mapping or (2) by choosing from the "preferred" model. This is covered by GMR 8 (Value preference).

Hence, given the input mapping, our algorithm provides a superset of PROMPT's merge functionality.

A similar tool is provided by the Chimæra system [1], part the Ontolingua Server [13] from Stanford's Knowledge Systems Laboratory. However, their tool concentrates almost entirely on the problem of Match rather than the problem of Merge. Specifically, their goal was to build a tool that "focuses the attention of the editor on particular portions of the ontology that are semantically interconnected and in need of repair or further merging." After discovering parts of an ontology (model) that need further merging, their algorithm operates much like ours; the two objects that have been equated are now combined into one object and their properties and relationships with other objects must be combined.

8.2.4 Object-Oriented Programming Languages

There are two aspects of programming languages that can benefit from a merge operator: multiple inheritance and paradigms where classes are combined to provide more information about a topic, such as subject-oriented programming. In multiple inheritance, when a class C inherits from more than one class, the resulting members of C are essentially the union of the members of its parents. A conflict occurs if C inherits from classes A and B and both A and B have a member x with different definitions. In this case, the classes are said to be incompatible and an error is returned [9]. This corresponds in model management to the user being unable to find an acceptable mapping.

Subject-oriented programming [15, 24, 25] is a programming paradigm that focuses on *subjects* rather than *objects*. A subject is a description of a number of objects and operations from one point of view. To determine how the subjects interact, their components are merged. A similar notion is aspect-oriented programming [17].

The key to subject-oriented programming is that different users need different information and functionality for the same subject. For example, a car rental agency has very different needs than a department of motor vehicles; a car rental agency is interested in when the car is rented, and a department of motor vehicles is interested in who licenses the car. Neither of these is inherent in the notion of a car. Thus Ossher and Harrison introduce the notion of *subjects*. A subject is "a collection of state and behavior specifications reflecting a particular gestalt, a perception of the world at large, such as is seen by a particular application or tool" [15].

Each subject is composed of a number of the following types of objects:

- Classes: The classes that are defined or used by the subject. Classes contain both member variables and member functions. Some member variables may be private to a subject, that is they should not merged if the subject is merged.
- Operations: The signatures (i.e., function name, parameters, and return type) of the functions used by the subject.

- Mapping: A list of how to map the (operation, class) pairs to the functions that need to be executed when a subject's operations are called; more than one function may need to be computed in order to provide one operation for a subject.

An example of a series of subjects and how to compose them is modeling cars and drivers from [25], which is shown in Figure 16. Note that in the combined subject, CarRenting, the class of Renter has been augmented by the variable "license" since Renter in the subject Renting matched the class Driver in the subject DMV. Also note the mapping for the operation (Check, Renter), which initially only called Renter.Check() now calls both Renter.Check() and Driver.GoodDriver(). This is because information on whether a renter is a good prospect can be gained from both the information stored by the rental agency and in the information stored by the department of motor vehicles.

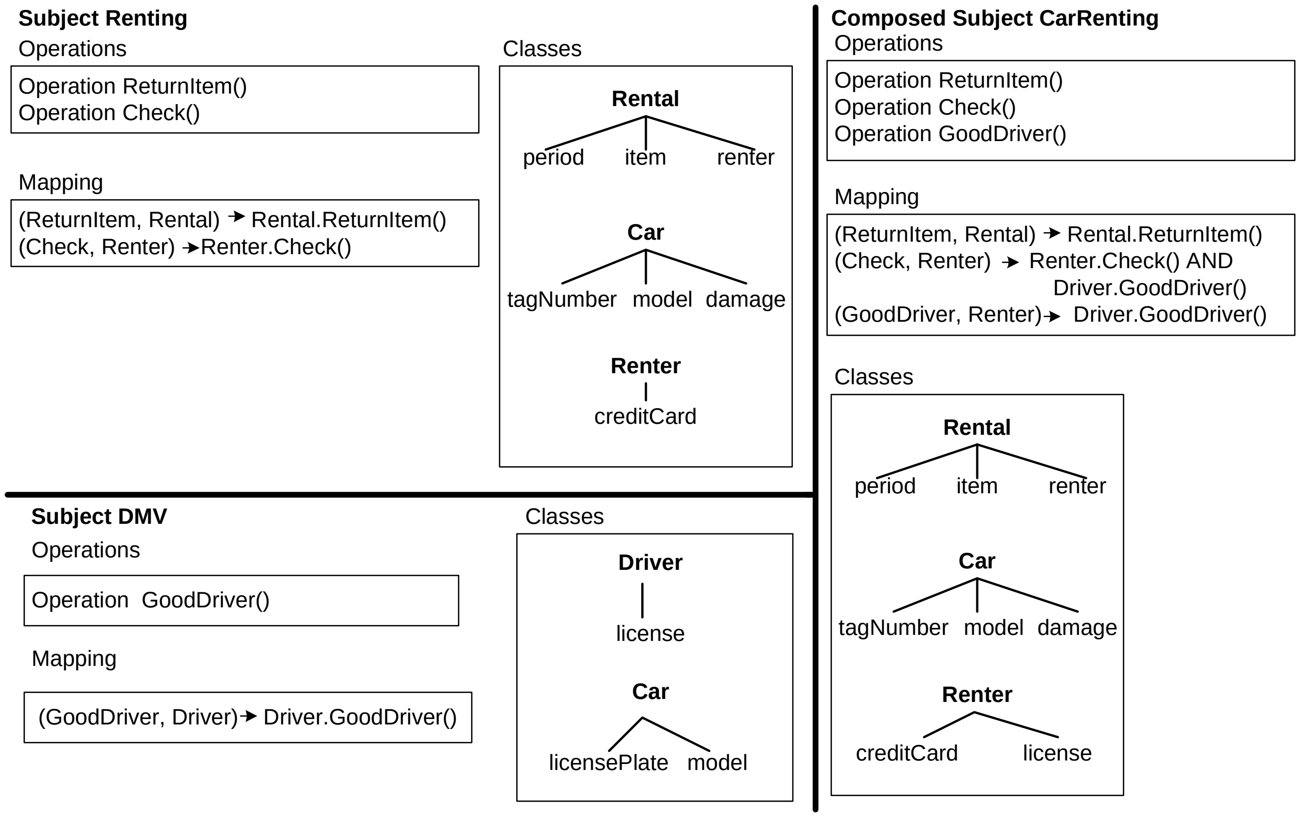


Figure 16: An example of subjects being composed [25]. Only member variables and not member functions are shown in the class diagrams. The Renter class in the Renting subject has been merged with the Driver class in the DMV subject to form the Renter subject in the composed subject CarRenting. The return type of the operations is not shown.

The subjects are specified in a normal object-oriented programming language, such as C++, and compiled to binaries. Then a "compositor program" is used to compose the subjects into the code that is actually executed; thus it is the operation of the compositor that we must examine in order to see how the subjects are merged and interact with one another. This is the operation that is most similar to merging in the context of model management. Like all other forms of Merge that we have discovered, they require input to tell whether the objects are related to one another.

To merge subjects, the classes, operations and mappings must be merged. We show below how each can be encoded in Vanilla and how model management operators can provide the same functionality as the subject-oriented programming merge.

We begin by showing how to merge classes. Classes, their member variables and member functions can all be encoded in Vanilla as elements. We model that a class has certain member variables or functions by saying that a Contains relationship exists between the class element and the member variable/function. For example, we would model the Rental class from the Renting subject shown in Figure 16 in Vanilla as shown in Figure 17. Since some variables are private to a subject (i.e., they should not be merged if the subject is merged), we use the Merge variant in Section 7.1 (merge only elements specifically mentioned in the mapping) to perform the merge.

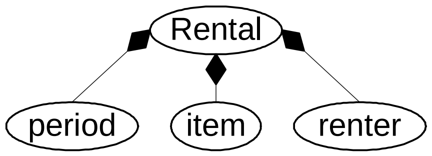


Figure 17: How to model the Rental class from the Renter subject shown in Figure 16 in Vanilla

Merging operations and mappings is very similar to merging classes, but the resolution of the property of a given operation or mapping is very different. We can encode operations and mappings in Vanilla similarly to how we encode classes. Each operation is encoded as an element that has sub-elements that specify (1) the parameters, and (2) the return type. Each mapping is encoded as an element that has a sub-element for (1) the subject – class pairs to be executed when an operation is called, and (2) which class and operation it belongs to. Each of these sub-elements may be broken into other sub-elements if necessary. In most merge algorithms that we have examined, the result of having conflicting information is that the property values of one of the elements should be taken and the other should be ignored. In the case of merging the mappings and operations in subject-oriented programming, the user must specify what the appropriate value should be, but it is quite often the combination of some, if not all, of the values of the properties. Ossher and Harrison provide a number of different resolutions for this. We would consider, however, that this is a problem that must be satisfied by the match (since it requires user intervention to discover the appropriate resolution) and hence would be encoded in the mapping in our algorithm.

9 Conclusions and Future Work

In this paper we defined the Merge operator for model merging, both generically and for a specific meta-meta-model, Vanilla. We defined and classified the conflicts that arise in combining two models and described when conflicts from different classes must be resolved. For conflicts that must be resolved in Merge, we gave resolution strategies, both for Vanilla and in general. We evaluated Merge by showing how Merge in Vanilla can be used to subsume some previous merging algorithms and by testing Merge on two large real-world ontologies.

We envision several future directions. The first involves showing that the Merge result, when applied to models and mappings that are templates for instances, has an appropriate interpretation on instances. This will demonstrate the usefulness of Merge in specific applications such as data integration and view integration [21, 31].

In some of our experiments, we encountered a complex structure in one model that is similar to a complex structure in another model. But rather than relating the elements by a complex expression, we wanted to relate individual elements but only in the context of that complex structure. An open question is how best to express such similarities and exploit them.

Finally, we would like to see a model-driven implementation of the EnforceConstraints operator that we proposed in Section 3.2.

Acknowledgements

We would like to thank Alon Halevy and Renée Miller for their continuing collaborations and Michael Ernst, Zack Ives, Peter Mork, and Steve Wolfman for their comments on earlier drafts of this paper. This work is partially funded by a Microsoft Research Graduate Fellowship.

References

1. Chimæra documentation, Knowledge Systems Laboratory at Stanford University.
2. Balasubramaniam, S. and Pierce, B.C., What is a File Synchronizer? MOBICOM, 1998, 98-108.
3. Batini, C., Lenzerini, M. and Navathe, S.B. A Comparative Analysis of Methodologies for Database Schema Integration. Computing Surveys, 18(4). 323-364.
4. Beeri, C. and Milo, T., Schemas for Integration and Translation of Structured and Semi-Structured Data. ICDT, 1999, 296-313.
5. Bergamaschi, S., Castano, S. and Vincini, M. Semantic Integration of Semistructured and Structured Data Sources. SIGMOD Record, 28 (1). 54-59.
6. Bernstein, P.A., Applying Model Management to Classical Meta Data Problems. CIDR, 2003, 209-220.
7. Bernstein, P.A., Halevy, A.Y. and Pottinger, R.A. A Vision of Management of Complex Models. SIGMOD Record, 29 (4). 55-63.
8. Biskup, J. and Convent, B., A formal view integration method. SIGMOD, 1986, 398-407.
9. Bracha, G. and Lindstrom, G., Modularity Meets Inheritance. Computer Languages Conference, 1992, 282-290.

10. Buneman, P., Davidson, S.B. and Kosky, A., Theoretical Aspects of Schema Merging. EDBT, 1992, 152-167.
11. Calvanese, D., Giacomo, G.D., Lenzerini, M., Nardi, D. and Rosati. Schema and Data Integration Methodology for DWQ, 1998.
12. Doan, A., Domingos, P. and Halevy, A., Reconciling Schemas of Disparate Data Sources: A Machine Learning Approach. SIGMOD, 2001.
13. Farquhar, A., Fikes, R. and Rice, J. The Ontolingua Server: A Tool for Collaborative Ontology Construction, 1996.
14. Guarino, N., Semantic Matching: Formal Ontological Distinctions for information Organization, Extraction, and Integration. Summer School on Information Extraction, 1997.
15. Harrison, W. and Ossher, H., Subject-Oriented Programming (A Critique of Pure Objects). OOPSLA, 1993.
16. Hernández, M.A., Miller, R.J. and Haas, L.M., Clio: A Semi-Automatic Tool For Schema Mapping. SIGMOD, 2001.
17. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M. and Irwin, J., Aspect-Oriented Progamming. ECOOP, 1997.
18. Larson, J.A., Navathe, S.B. and Elmasri, R. A Theory of Attribute Equivalence in Databases with Application to Schema Integration. Trans. on Soft. Eng., 15(4). 449-463.
19. Madhavan, J., Bernstein, P.A. and Rahm, E., Generic Schema Matching with Cupid. VLDB, 2001, 49-58.
20. Melnik, S., Rahm, E. and Bernstein, P.A., Rondo: A Programming Platform for Generic Model Management. SIGMOD, 2003, To Appear.
21. Motro, A. Superviews: Virtual Integration of Multiple Databases. Trans. on Soft. Eng., SE-13(7). 785-798.
22. Noy, N.F. and Musen, M.A., PROMPT: Algorithm and Tool for Ontology Merging and Alignment. AAAI, 2000.
23. Noy, N.F. and Musen, M.A., SMART: Automated Support for Ontology Merging and Alignment. Banff Workshop on Knowledge Acquisition, Modeling, and Management, 1999.
24. Ossher, H. and Harrison, W., Combination of Inheritance Hierarchies. OOPSLA, 1992.
25. Ossher, H., Kaplan, M., Katz, A., Harrison, W. and Kruskal, V. Specifying Subject-Oriented Composition. Theory and Practice of Object Systems, 2(3). 179-202.
26. Rector, A., Gangemi, A., Galeazzi, E., Glowinski, A. and Rossi-Mori, A., The GALEN CORE Model Schemata for Anatomy: Towards a re-usable application-independent model of medical concepts. The Twelfth International Congress of the European Federation for Medical Informatics, 1994.
27. Rosse, C., Shapiro, L.G. and Brinkley, J.F., The digital anatomist foundational model: principles for defining and structuring its concept domain. AMIA, 1998, 820-824.
28. Shu, N.C., Housel, B.C. and Lum, V.Y. CONVERT: A High Level Translation Definition Language for Data Conversion. Communications of the ACM, 18(10). 557-567.
29. Song, W.W., Johannesson, P. and Bubenko, J.A., Jr. Semantic Similarity Relations in Schema Integration. Data Knowledge and Eng., 19(1). 65-97.
30. Spaccapietra, S. and Parent, C. View Integration: A Step Forward in Solving Structural Conflicts. TKDE, 6(2).
31. Ullman, J.D., Information Integration Using Logical Views. ICDT, 1997, 19-40.

A. Compose

The Compose operator takes a mapping, Map_{AB}, between models A and B and a mapping, Map_{BC}, between models B and C and returns, Map_{AC}, the composed mapping between A and C. We define Compose based on the definition of right composition (i.e., composition driven by the right hand mapping) in [6].

Let m be a mapping element in mapping Map_{AB} between models A and B. Define domain(m) (resp., range(m)) to be all elements e such that $e \in A$ (resp. $e \in B$) and M(m, e). For each element e in the domain of each mapping element m in Map_{BC}, Compose must identify the mapping elements in Map_{AB} that provide input to e. We compose each element m_{BC} in map_{BC} with the union of all elements $m_{AB} = m_{AB1}, \ldots, m_{ABn}$ in map_{AB} where $range(m_{AB}) \cap domain(m_{BC}) \neq \emptyset$.

Given this decision, we define the composition map_{AC} of map_{AB} and map_{BC} constructively as follows:

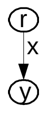
1. (Copy) Create a copy map_{AC} of map_{BC}. Note that map_{AC} and map_{BC} have corresponding mapping relationships to B and C and, therefore, the same domains and ranges.
2. (Precompute Input) For each object m_{AC} in map_{AC}, let Input(m_{AC}) be the set of all elements m_{AB} in map_{AB} such that $range(m_{AB}) \cap domain(m_{BC}) \neq \emptyset$.
3. (Define domains) For each m_{AC} in map_{AC},
   a. If $\bigcup_{m_{ABi} \in Input(m_{AC})} range(m_{ABi}) \supseteq domain(m_{AC})$, then set domain($m_{AC}$) = $\bigcup_{m_{ABi} \in Input(m_{AC})} domain(m_{ABi})$.

b. Else if m_{AC} is not needed as a support element[2] (because none of its descendants satisfies (3a)), then delete it, else set domain(m_{AC}) = range(m_{AC}) = ∅.

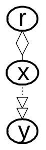
Step 3 defines the domain of each object m_{AC} in map_{AC}. Input(m_{AC}) is the set of all objects in map_{AB} whose range intersects the domain of m_{AC}. If the union of the ranges of Input(m_{AC}) contains the domain of m_{AC}, then the union of the domains of Input(m_{AC}) becomes the domain of m_{AC}. Otherwise, m_{AC} is not in the composition, so it is either deleted (if it is not a support object, required to maintain the well-formed-ness of map_{AC}), or its domain and range are cleared (since it does not compose with objects in map_{AB}).

B. Our Modifications to the BDK Algorithm

BDK combine the representation of our Has-a and Type-of relationships into one relationship. They represent the fact that an element r Has-a element x of type y by an arrow from r to y with the label x. The different representations are shown in Figure 18 (a single Has-a relationship) and in Figure 19 (a violation of the one type restriction). Although they represent both the Vanilla Has-a and Type-of relationships with a single relationship, the BDK algorithm only involves duplicate element types, not duplicate containers, hence our modification involves only transforming the Vanilla Type-of relationships rather than the associated Has-a and Contains relationships.

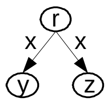


(a)

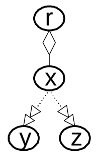


(b)

Figure 18: Modeling Has-a and Type-of relationships. (a) In the BDK meta-meta-model (b) In Vanilla



(a)



(b)

Figure 19: A violation of the one type restriction. (a) In the BDK meta-meta-model (b) In Vanilla

Applying the BDK algorithm to a model M expressed in Vanilla requires the following conceptual steps:

1. Apply all implied relationships (listed in Section 4.1) that can lead to the inclusion of additional Type-of or Is-a relationships.
2. Apply a transformation T1(M)→N to transform M into a model N in BDK's meta-meta-model. This transformation operates in the following fashion:
   a. Each element m of M becomes a node n in N, where the label of n is the value of the ID property of m.
   b. Each Vanilla Is-a relationship between two elements in M becomes a BDK Is-a relationship between the corresponding nodes in N.
   c. For each Vanilla Type-of relationship in M, a BDK Has-a relationship with label "t" is created between corresponding nodes in N. Note that the label is unimportant except that it must be the same for the BDK algorithm to function correctly.
3. Run the BDK algorithm to change weak schemas (those that do not obey the one type restriction) into strong schemas (those that obey the one type restriction).
4. Remove all Type-of and Is-a relationships from M (i.e., all relationships imported into N are removed)
5. Apply a transformation T2(N)→M to add into M:
   a. All nodes from N that do not correspond to elements in M (i.e., nodes that were created to help resolve one type conflicts).
   b. All relationships in N (i.e., all relationships that were originally imported into N plus all of the changes and additions to resolve one type conflicts). All Is-a relationships are added directly between corresponding elements since the Is-a relationships are the same in BDK and in Vanilla. All BDK Has-a relationships are transformed into Vanilla Type-of relationships, and their labels are discarded (since we specified that these would all be equal to the arbitrary choice "t", no information is lost in this transformation)
6. All implied relationships are removed.

[2] A support element is an element needed to support the structural integrity of the model (i.e., an element needed to ensure that the result is a model)

This algorithm guarantees:

1. All relationships originally in M are retained at least implicitly. All relationships that are not Is-a or Type-of relationships are retained. All Is-a and Type-of relationships are imported from N. Since the BDK transformation only changes the relationships based on the implication rule "If T(q, r) and I(r, s) then T(q, s)," which exists in Vanilla and makes the same guarantee of retaining all relationships at least implicitly, this is retained.
2. No element in M has more than one type. The BDK algorithm ensures that this is true for Has-a relationships in N. No Type-of relationships are included in M other than those in N, and T2's transformation between Has-a and Type-of relationships cannot violate this. The final step of removing implied relationships cannot violate this since no new relationships are added.
3. The associativity and commutativity properties of BDK are retained. Both T1 and T2 are entirely order independent. Note that in order for the conflict resolution to be associative and commutative, the algorithm must be run only *once* at the end of a series of merges (see [10] for an explanation as to why).

C. Three-Way Merge Algorithm

1. Map_{OA} = Match(O, A) (can be automatic from History properties)
2. Map_{OB} = Match(O, B) (can be automatic from History properties)
3. $Map_{OA'}$ = Apply(Map_{OA}) such that if $e \in Map_{OA}$ if domain(e) is identical to range(e), then delete e (we are capturing the things changed in A)
4. $Map_{OB'}$ = Apply(Map_{OB}) such that if $e \in Map_{OB}$ if domain(e) is identical to range(e), then delete e (we are capturing the things changed in B)
5. $Changed_A$ = range($Map_{OA'}$) (the things changed in A)
6. $Changed_B$ = range($Map_{OB'}$) (the things changed in B)
7. Map_{ChBChA} = Match($Changed_A$, $Changed_B$)
8. A′ = Diff($Changed_A$, $Changed_B$, Map_{ChAChB})
   (A′ represents the things changed in A that were not changed in B, and mutatis mutandis for B′ below)
9. B′ = Diff($Changed_B$, $Changed_A$, Map_{ChBChA})
10. Map_{AB} = Match(A,B) (according to OIDs)
11. G = Merge(A, Map_{AB}, B)
12. $Map_{GA'}$ =Match(G,A′)
13. GA = Merge(G, $Map_{GA'}$, A′) with preference for A′
14. $Map_{GA'B'}$ =Match(GA′,B′)
15. GAB = Merge(GA′, $Map_{GA'B'}$, B′) with preference for B′
    (GAB represents the full merge with a preference for those things that have changed in either A or B but not both)
16. $Deleted_A$ = Diff(O,A,Map_{OA})
17. $Deleted_B$ = Diff(O, B, Map_{OB})
18. $Map_{DeletedAChangedB}$ = Match($Deleted_A$, $Changed_B$)
19. $Map_{DeletedBChangedA}$ = Match($Deleted_B$, $Changed_A$)
20. $ShouldDelete_A$ = Diff($Deleted_A$, $Changed_B$, $Map_{DeletedAChangedB}$)
21. $ShouldDelete_B$ = Diff($Deleted_B$, $Changed_A$, $Map_{DeletedBChangedA}$)
22. Map_{GABSDA} = Match(GAB, $ShouldDelete_A$)
23. GABSDA = Diff(GAB, $ShouldDelete_A$, Map_{GABSDA})
24. $Map_{GABSDASDB}$ = Match(GABSDA, $ShouldDelete_B$)
25. Final result = Diff(GABSDA, $ShouldDelete_B$, $Map_{GABSDASDB}$)

D. The PROMPT Algorithm

A. The user performs setup by loading the models, A and B, and specifying some options. If the operation is merge, the result model C is initialized to be a new model with a new root and A and B as that root's children.

B. PROMPT generates an initial list of suggestions, based largely on content or syntactic information. It examines the objects, but not the structural information (i.e., the position of the objects or their participation in specific relationships as represented by relationships between the objects).

   If the operation is merge:

   a. For each pair of objects $a \in A$ and $a \in B$ with identical names PROMPT either merges the a and b in C or removes either a or b from C.

   b. For each pair of objects a ∈ A and b ∈ B with linguistically similar names a link is created between them in C (with a lower degree of confidence than if the names were identical). This means that both a and b are still in C, but PROMPT suggests that they may need to be merged by adding them to the ToDo list.

      If the operation is match and the user has tagged one model (say, A) as more general during setup, then PROMPT assumes that the objects in the less general model (say, B) should be linked in as sub-objects of the objects in A. If there is a top-level object, t, in B, with the same name as an object in A, then the two objects are merged in C. Otherwise finding a parent object for t is added to the ToDo list.

C. The user selects and performs an operation such as merging an object or resolving an item on the ToDo or Conflict lists.
D. PROMPT performs any automatic updates that it can and creates new suggestions. It has the ability to:
   c. Execute any changes automatically determined as necessary by PROMPT.
   d. Add any conflicts caused by the user's actions in step 3 to the Conflicts list
   e. Add to the ToDo list any other suggested operations or make new suggestions based on linguistic similarity or structural stability.
E. Steps 3 and 4 are repeated until the ontologies are completely merged or matched.