A Manifesto for Model Merging

Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, Mehrdad Sabetzadeh
Department of Computer Science, University of Toronto
Toronto, ON M5S 3G4, Canada.
Email: {gbrunet,chechik,sme,shiva,nn,mehrdad}@cs.toronto.edu

Abstract

If a modeling task is distributed, it will frequently be necessary to merge *models developed by different team members. Existing approaches to model merging make assumptions about the types of model to be merged, and the nature of the relationship between them. This makes it hard to compare approaches. In this paper, we present a manifesto for research on model merging. We propose a framework for comparing different approaches to merging, by treating merge as an algebraic operator over models and model relationships. We specify the algebraic properties of an idealized* ***merge*** *operator, as well as related operators such as* ***match****,* ***diff****,* ***split****, and* ***slice****. We then show how our framework can be used to compare existing approaches by applying it to two of our own research projects on model merging. We show how this analysis permits a detailed comparison of approaches, reveals the key features of each, and identifies weaknesses that require further research. Most importantly, the framework emphasizes the need to make explicit all assumptions about the relationships between models, and indeed to treat* model relationships *as first class objects.*

1 Introduction

In model-based development, models are usually constructed and manipulated by distributed teams, each working on a partial view of the overall system. Model management involves keeping track of the relationships between the various models, and, from time to time, combining information from several models into a single model. We call this *model merging*.

A number of factors make model merging complicated. The modelers may have used different vocabularies, or applied a shared vocabulary inconsistently. Models may overlap, in that they refer to the same concepts in the problem or solution domains, but the overlapping concepts may be presented differently in each model, and the models may contradict one another.

The problem of model merging has been studied in many domains: schemata of several independently-developed databases [2, 16, 11], static and dynamic UML models [23, 4], web services [10], program variants [7], requirements models [18, 19], and many varieties of reactive systems [9, 22, 3, 6].

The modeling formalisms and the underlying assumptions in these approaches differ significantly. For example, most of these approaches require that only consistent models be merged, while some tolerate inconsistency. In some cases, the goal is to show *differences* between the models rather than to put them together [23].

The approaches disagree on the treatment of information not explicitly present in the model. For example, [2, 11, 6, 3] assume that omitted information is "possible" (e.g., an equation "x = 3" does not constrain the value of y), whereas most state-machine approaches assume that omitted information signifies prohibited behaviours; [22] takes a hybrid approach.

These approaches further differ in whether they handle heterogeneous modeling languages. Typically, models are assumed to be homogeneous. For example, the majority of UML merging approaches first translate them into the underlying unified language and only then merge them. [6] is a notable exception which allows merging scenarios and state machines directly.

Finally, the approaches range in the level of sophistication in matching and identifying similar information. In software engineering, most approaches assume that entities are the same when they have the same name or id (or are derivatives of entities with the same name or id); others, e.g., [18], support the use of an explicit ontology, or a thesaurus, relating entities from each model. In the database work, schema integration approaches implement sophisticated matching algorithms [17].

In this paper, we propose a unifying framework for understanding the model merging problem. Our intent is to facilitate comparison and extension of existing approaches, and to suggest a research agenda for future work.

Our framework is based on two observations about model merging:

- In practice, model merging is an exploratory process, in which the goal is to discover the exact nature of the relationship between models, as much as to combine them. Hence, we need to be explicit about these relationships, and need to provide the modelers with a way of stating them precisely.

- Model merging is facilitated by a number of related operations on models, such as comparing models, checking their consistency, and finding matches between them. We can characterize each of these as algebraic operations over models and model relationships, in order to understand their properties.

We begin the paper with a characterization of a set of useful model management operators (Section 2). We then describe the algebraic properties of the ideal merge operator (Section 3). Afterwards, we show how this framework can be used to analyze particular approaches to model merging, using two of our own projects on model merging (Section 4). The first of these is a general approach to merging conceptual models based on structural mappings between them. The second is a technique for weaving state machine models by finding a common refinement. We discuss the merge framework in Section 5 and conclude the paper with an outline of the future research agenda for model merging (Section 6).

2 Model Management Operators

In this section, we present a basic set of operators for global model management, in terms of the objects they manipulate. The basic data types in our definitions include *models*, *relationships* between models, and *properties* of models. At this point, we consider arbitrary models, but note that particular implementations of these operators might be defined only for certain classes of model.

We begin with the *merge* operator:

$$\textbf{merge} : \textit{model} \times \textit{model} \times \textit{relationship} \rightarrow \textit{model}$$

We assume the *models* to be merged are of the same type, or of compatible sub-types within the type hierarchy. A *relationship* must specify how the models as a whole relate to one another, as well as any specific mappings between elements within the models, for example, any mappings between the vocabularies of the models. The key idea is that there may be many different ways of putting together an arbitrary pair of models; the relationship specifies which of these is desired. For example, when composing behavioural models (e.g. state machines), the behaviours might be composed in parallel, sequentially, or as restrictions or extensions of one another.

In our past work on merging as well as in reviewing the relevant literature, we have found a number of supporting operations on models to be useful. For example:

$$\textbf{match} : \textit{model} \times \textit{model} \rightarrow \textit{relationship}$$

Match is sometimes also referred to as *mapping*. This operation is used to find commonalities between models. The resulting relationship can be used as a basis for model merging. The result of *match* may be under-determined, in that there may be many possible relationships between the models. In this case, *match* can be used to generate candidate relationships for exploratory merging.

$$\textbf{diff} : \textit{model} \times \textit{model} \rightarrow \textit{transformation}$$

Match returns a relationship, whereas *diff* returns a sequence of edit actions needed to transform one model into another. The obvious example is the UNIX utility diff, which performs this operation on text files. Note that *diff* is not commutative: the transformation is *from* the first model *to* the second. In some cases, we may only be interested in the length of the transformation, i.e., the edit distance between the two models.

$$\textbf{split} : \textit{model} \rightarrow \textit{model} \times \textit{model} \times \textit{relationship}$$

Split produces a *partition*, and can be viewed as the inverse operation of *merge*. The resulting *relationship* of *split* can be further understood as *interfaces* between partitioned models. *Split* produces compatible models.

$$\textbf{slice} : \textit{model} \times \textit{criterion} \rightarrow \textit{model}$$

Slice produces a *projection*, or a partial view of a model, based on a *criterion*. Therefore, the resulting model contains every piece of the input model that satisfies the criterion, and thus can be used to build *aspect(s)* of the input model.

Because some types of merge require the input models to satisfy certain (behavioural and/or structural) properties, as well as consistency, we may also need:

$$\textbf{check-property} : \textit{model} \times \textit{property} \rightarrow \textit{truth-value}$$
$$\textbf{is-consistent} : \textit{model} \times \textit{model} \times \textit{relationship} \rightarrow \textit{truth-value}$$

These operations can be used to check the models prior to merging. Note that consistency checking can only be performed with respect to an intended relationship between two models [15].

Finally, we identify two more operators that are useful for global model management:

$$\textbf{patch} : \textit{model} \times \textit{transformation} \rightarrow \textit{model}$$
$$\textbf{propagate} : \textit{transformation} \times \textit{model} \times \textit{model} \times \textit{relationship} \rightarrow \textit{model}$$

Patch applies a transformation, such as that returned by *diff*, to a model to generate a revised model. *Propagate* takes a transformation intended for one model, and applies a corresponding transformation to a second model, where the two models have a defined relationship between them. The intent is to propagate edits to related models to keep them consistent.

3 Algebraic Properties

Having characterized merge and its related operators in terms of the objects they manipulate, we now specify some of the properties we might expect of these operators. For example, does the order of merging various models matter? Is merge monotonic? We describe the properties in algebraic terms, and then discuss their practical utility.

In the following, $m_1, m_2, \ldots$, are assumed to be models of the same type (or compatible sub-types), and $r_{i,j}$ is a relationship between m_i and m_j. We drop the subscripts when the context is clear.

1. Idempotency:
   $\mathit{merge}(m_1, m_1, \mathit{match}(m_1, m_1)) = m_1$
   Merging a model with itself should return the same model, assuming that the given relationship completely maps the model onto itself. Hence, we assume that matching a model with itself should produce such an identity relationship. In practice, this property may be too strong, as we may be willing to accept a result that is isomorphic, rather than identical, to the original.

2. Commutativity:
   $\mathit{merge}(m_1, m_2, r) = \mathit{merge}(m_2, m_1, r)$
   This simply states that, for a given relationship, it should not matter which order the models are presented in.

3. Associativity:
   $\mathit{merge}(\mathit{merge}(m_1, m_2, r_{1,2}), m_3, r_{(1,2),3}) =$
   $\mathit{merge}(m_1, \mathit{merge}(m_2, m_3, r_{2,3}), r_{1,(2,3)})$
   If the merge operation is associative, then generalization to more than two models can be achieved merely by repeated merges, in any order. In practice, such a generalization is complicated by the need to specify the relationship at each step. It may be more practical to define a general merge operator that works for sets of models and relationships.

4. Inverses:
   $\mathit{split}(\mathit{merge}(m_1, m_2, r)) = (m_1, m_2, r)$
   merge and *split* are inverses of one another.

5. Monotonicity:
   $m_1 \preceq m_1' \wedge m_2 \preceq m_2' \Rightarrow$
   $\mathit{merge}(m_1, m_2, r) \preceq \mathit{merge}(m_1', m_2', r')$
   where $\preceq$ is some pre-order defined over models (e.g., refinement and trace equivalence [12]) that is logically characterized by the *properties* that are preserved by merge. The idea here is that the merge operands m_1 and m_2 can be evolved independently (to m_1' and m_2'), such that the merge of the evolved operands is guaranteed to be an evolution of the original merge. In practice, this means that the merge of the evolved models preserves the desired *properties* of the original merge. This facilitates component-based reasoning with respect to merge.

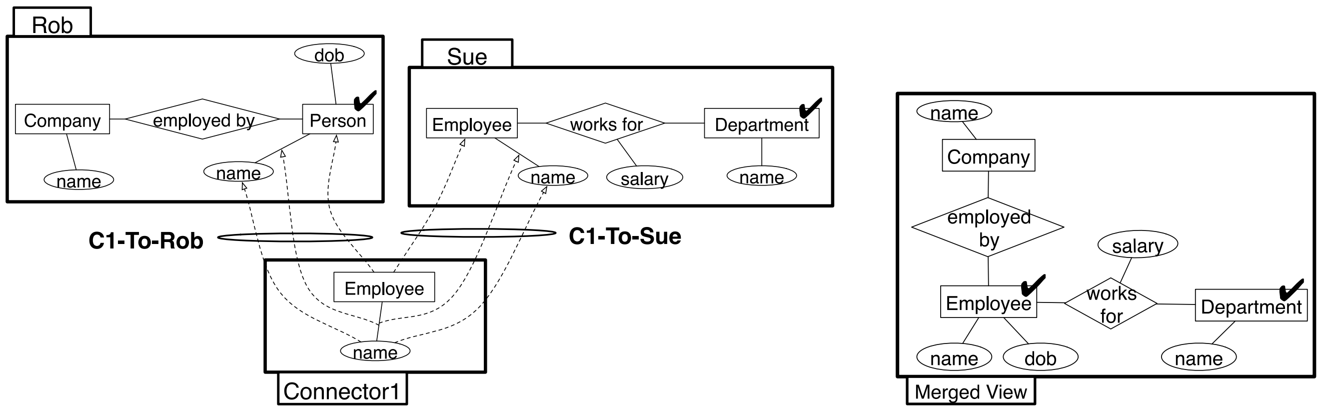
6. Totality:
   $\forall m_1, m_2 \in \mathit{model} \cdot \mathit{merge}(m_1, m_2, r) \in \mathit{model}$
   This property is of particular importance, as its satisfaction means that the merge operation is well-defined for any pair of models, whether or not they satisfy conditions such as consistency. If this property holds, it means we can generate a merge, even for inconsistent models. In some cases this is preferable to repairing the inconsistency first. In particular, with tools for reasoning in the presence of inconsistency, the merged model can be used as a basis for exploration and negotiation.

Note that the above algebraic properties are "ideal". Satisfying a reasonable subset of these (typically as a result of simplifying assumptions or limitations of the specific modeling notation) still facilitates a useful merge operation in practice, as we illustrate below.

4 Examples

To illustrate our global model management framework, we instantiate it to two different domains: In Section 4.1 we show how structural models represented as entity relationship diagrams (ERDs) can be merged. In Section 4.2, we discuss merging behavioural models described as state machines. A major aspect of our previous work on model merging has been dealing with partiality and inconsistency. To capture partiality and inconsistency, we use multi-valued logics. The different values of our logics represent different degrees of stakeholders' knowledge. In the examples of Sections 4.1 and 4.2, a 3-valued logic [8] is used. The truth values of this logic are represented and interpreted differently in our examples: In the ERD example, the truth-values are drawn from the set {✔, **!**, ✘ }: ✔ is used when an element is conclusively *appropriate*; ✘ is used when an element is conclusively *inappropriate*; and **!** is used when an element is *proposed* but is not yet known to be appropriate (or inappropriate) for sure.

In the state machine example, the truth-values are denoted by the set {`t`, **!**, `f` }: `t` is interpreted as *true* and



(a) Interrelating the perspectives

(b) The merged model

Figure 1. An example of merging ERDs.

is used for elements explicitly included in a model; `f` is interpreted as *false* and is used for elements explicitly excluded from a model; and **!** is interpreted as *unknown* and is used for elements whose value may either be *true* or *false*. The reason why the interpretations of the truth values differ in our structural and behavioural examples is because for structural models an *open world semantics* is implicitly assumed. This is in contrast to behavioural models for which a *closed world semantics* is typically used.

4.1 Merging Entity-Relationship Models

In this section, we briefly sketch out our structural merging approach described in [19] and illustrate it by walking through a simple requirements elicitation exercise where we consolidate requirements models originating from different human sources. Suppose Rob and Sue are gathering the requirements for a payroll database. An analyst, Jack, will help them through the elicitation process and identification of correspondences between their models. Stakeholders' perspectives are captured by ERDs.

The first step is for Rob and Sue to describe their initial perspectives (models **Rob** and **Sue** in Figure 1(a)). They use the 3-valued truth set described earlier in the paper for stating their level of confidence about each element in their models. For convenience, "proposed" (**!**) is treated as a default annotation for all model elements, and only the remaining values are shown. We do not use ✘ in this example.

Inputs. We use two main abstractions for describing a merge: *models* and *embeddings*. A *model* is simply a graphical representation of a set of related concepts, here ERDs. An *embedding* expresses an admissible way to incorporate a model into another. Embeddings preserve the graphical structure of models and are described by graph homomorphisms. The input to the merge process is a set of models and a set of embeddings between them.

The Merge Process. Our structural merging algorithm is based on a category-theoretic concept called *colimit* [1]. Colimits provide an abstract and yet very powerful machinery for merging [5]. Intuitively, computing the colimit yields a new model combining all models w.r.t. their correspondences as described by the embeddings.

Below, we illustrate colimits in our elicitation example: Once Rob and Sue provide their models, Jack will merge them to create a unified schema. To do the merge, Jack needs to identify the correspondences between Sue's and Rob's models: Employee in Sue's model is likely to be the same entity as Person in Rob's. Consequently, the name attribute of Employee would be the same as that of Person.

To describe these correspondences, Jack creates a new model, **Connector1** (shown in Figure 1(a)), containing only the parts that are in common between Rob's and Sue's models. He then chooses appropriate names for the elements in **Connector1** and specifies how the model is embedded into each of the stakeholders' models using two embeddings **C1-To-Rob** and **C1-To-Sue**. We usually refer to shared models as *connectors* because they are used to describe correspondences between other models. Notice that even if Rob used the term Employee instead of Person in his model, defining a connector would still be necessary because our merge framework does not rely on naming conventions to describe the desired unifications – all correspondences must be identified explicitly prior to the merge operation. For the set of interconnected models in Figure 1(a), computing the colimit results in a new ERD expressing the union of Rob's and Sue's perspective such that their overlap, described by

Connector1, is included only once.

The result of the merge operation is shown in Figure 1(b). For naming the elements of this model, we have assumed that the naming choices in the connector (which happen to favor Sue in this particular example) take precedence over those in the stakeholders' models.

Relationships. The way we described correspondences between two models implies a certain notion of inter-model relationships: A *relationship* between a pair of models A and B is a model C together with embeddings $f : C \to A$ and $g : C \to B$ describing how C is represented in each A and B. For example, **Connector1**, **C1-To-Rob** and **C1-To-Sue** in Figure 1(a) describe a relationship between Rob's and Sue's models.

The Match Operator. The output of the *match* operator in this approach would be a relationship as described above. In our example, the task of identifying matches is not automated but rather left to an analyst.

The Split Operator. *split* can easily be implemented in terms of the traceability information stored in merges. The details of our traceability framework have been described in [20].

Check-property. We do not assume any particular semantics for our structural models. *check property* depends on domain semantics and is not addressed in our work.

Consistency. Our structural merges tolerate partiality and inconsistency. A complete discussion on this subject is out of the scope of this paper. See [19] for details.

Algebraic Properties. Idempotency, commutativity, and associativity immediately follow when colimits are used for merging. Our merges have inverses when proper traceability information is stored in them [20]. If we interpret $\preceq$ as model inclusion, monotonicity also holds. Embeddings preserve syntactic constraints, so *merge* is total. However, since embeddings do not necessarily preserve semantics, the resulting merges may not be semantically sound.

Other Operators. We do not address *diff*, *slice*, *patch*, and *propagate*.

4.2 Weaving State Machines

In this section, we give an outline of our behavioural merging approach [22, 13] and apply it to merging two variant interpretations of the photo-taking functionality of a camera.

Inputs. Each variant is described as a state machine consisting of a set of states, a set of transitions between states, and a set of variables whose values vary from state to state. Each variable is assigned one of the three values: `t`, **!**, or `f`. By convention, an atomic proposition in positive form is `t`, and a negated proposition is `f`. Unknown propositions are annotated by **!**.

The informal specification of the photo-taking functionality is as follows:

> To take a photo, a user needs to press the *shutter* button half-way. When *focus* is achieved, the shutter button can be pressed completely to take a picture. Under low-light conditions, the built-in *flash* should fire automatically.

Two possible specifications of this functionality, $\mathbf{CM_1}$ and $\mathbf{CM_2}$, are shown in Figure 2. The goal of $\mathbf{CM_1}$ is to specify the focusing feature and the behaviour of the camera's shutter. In state s_0, the shutter is closed and the focus is not yet achieved; in s_1, the focus is achieved; and in s_2, the shutter opens and a photo is taken. Model $\mathbf{CM_2}$ additionally describes the built-in flash, which is disabled in states t_0 and t_1. In t_2, the camera opens its shutter and, depending on the light intensity, the flash is either fired or remains disabled.

The Merge Process. To merge models, we first unify their variable sets and address the resulting incompleteness by setting missing variables to **!**. For example, when the set of variables of $\mathbf{CM_1}$ is lifted to the unified set of variables, we simply set the missing variable `Flashing` to **!** in all of the states of $\mathbf{CM_1}$. The result is the model $\mathbf{CM_1'}$, shown in Figure 2.

The intuition we wish to capture by behavioural merging is that of combining partial knowledge coming from individual models while preserving all of their agreements. The notion of *common refinement* underlies this intuition as it captures the "more complete than" relation between two incomplete models [22, 13]. Models can have more than one common refinement. We are always interested in the least one. In order to compute the least common refinement, i.e., the merge, of two state machines, we first compute a *relationship* between the states of the two models. This relationship effectively allows us to describe the correspondences between two models and compute their merge. For our example, the merge of $\mathbf{CM_1}$ and $\mathbf{CM_2}$ is $\mathbf{CM_3}$, shown in Figure 2.

Relationships. In this approach a *relationship* is a binary relation between the states of the input models. For example, the relationship between $\mathbf{CM_1}$ and $\mathbf{CM_2}$ is $\{(s_0, t_0), (s_1, t_1), (s_2, t_2), (s_2, t_3)\}$.

The Match Operator. The output of *match* is a relationship as described above. When the least common refinement exists, this relation is unique and can be computed automatically.

Check-property. We assume our state machines have

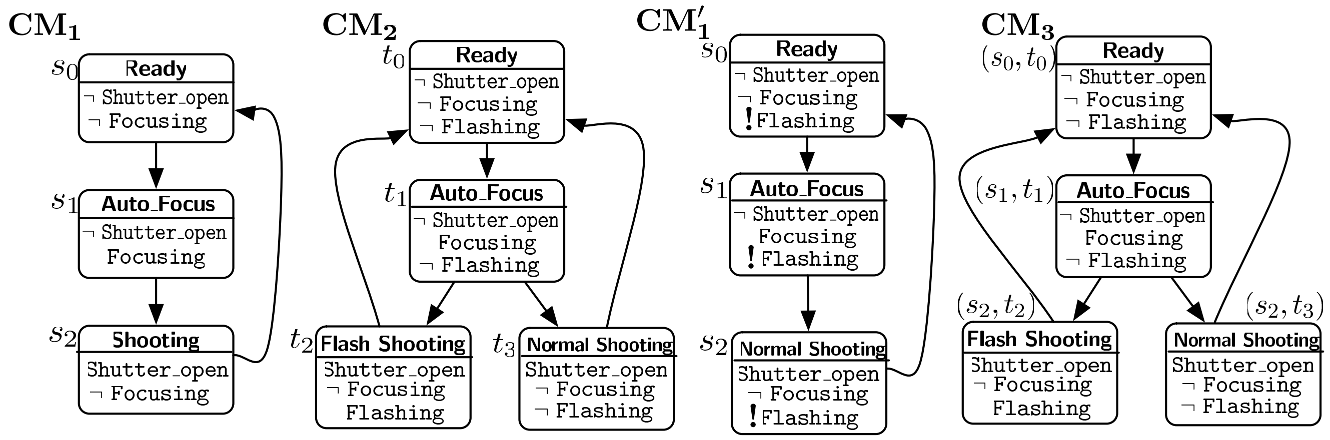


Figure 2. An example of merging state machines.

behavioural semantics. Our merge preserves model behaviours. This allows us to employ model checking techniques throughout the merge process. The usage of the *check-property* operator in our approach is elaborated in [13]. For example, the property "whenever focus is achieved, we can take a picture" is formalized by the CTL formula $AG(\mathsf{Focusing} \rightarrow EX\,\mathsf{Shutter\_open})$. This property holds in both $\mathbf{CM_1}$ and $\mathbf{CM_2}$ and therefore is satisfied by the merged model $\mathbf{CM_3}$ as well.

Consistency. Our approach tolerates partiality and can identify inconsistencies. We disallow merging inconsistent models. Further details can be found in [22, 13].

Algebraic Properties. Given a relationship between a pair of models, idempotency, commutativity, and associativity hold. Monotonicity of merges also follows if we interpret $\preceq$ as a refinement relation over models. Totality does not hold: only consistent models can be merged.

Other Operators and Properties. We do not address *diff*, *split*, *slice*, *patch*, and *propagate*.

5 Discussion

Our intent in this paper is to develop a framework for comparing different approaches to model merging. Such a framework will be useful if it provides a suitable vocabulary for discussing the key ideas in model merging, if it reveals key issues and assumptions in existing research projects, and provides a rich source of suggestions for future work.

Our list of model operators in Section 2 clearly distinguishes the different possible operations on models, and provides a standard vocabulary. The distinctions between the operators have already proved useful in discussing our own work on merging. For example, we had already identified the need to develop a *match* operator for our merge tools [14], but had not explicitly considered the use of other operators, and how they would interact with *merge*. A model development process might use *split* at some stage to divide a large model into separate workpieces, to be evolved independently. The relationship produced by *split* could then be used either to *propagate* edits from one model to another, or as the basis for an eventual re-*merge*. Alternatively, if we have models that originated from different sources, we might transfer information from one model to another by using *slice* to extract an aspect from one source model, and then *match* and *merge* to apply it to another.

A key insight gained by treating models and their operations as an algebra is the central role played by *relationships*, and the key questions of how relationships are discovered and represented. The relationships we have been using focus on mappings between model elements. In our work on merging behavioural models, the relationship is as simple as a list of pairs of states to be unified. In our work on merging structural models, the relationship is an overlap (expressed as a third model), together with information about how that overlap is manifested in each model. In both cases, we may have to capture differences of vocabulary, and preferences for which vocabulary to use in the merged model.

A comparison between these different ways of representing relationships reveals more assumptions about the nature of relationships between models. Our relationships express overlaps – concepts that are represented in both models, and must be unified when the models are merged. However, this implies a broader assumption about how the models as a whole relate to each other. In both approaches, the models are assumed to exhibit *complementarity* – each describes some concepts that are ignored in the other. Such an assumption may fit poorly with the usual semantics of the modeling languages. For example, state machine formalisms traditionally adopt a closed world semantics (nothing can occur, other than the stated transitions). By declaring two models to be complementary, we are overriding the closed world assumption. In some cases, we may want to treat some aspects of the models not as complementary, but rather as contradictory. For example, in UML, if two class models give different sets of attributes for a class that is to be unified, we may view this as a disagreement about the correct set of attributes, rather than as a complementarity. In most approaches to merging, the choice over whether a given relationship between model elements is a contradiction or a complementarity is implicitly determined by the model semantics. It would be better to specify this explicitly, and to consider how the stated relationship interacts with the model semantics.

The analysis of whether specific merge operators exhibit *totality* reveals a significant variability in how merge tools check and handle assumptions on their inputs. Some merge approaches assume no name clashes occur, or at least that there are no homonyms in the unified vocabulary of the models to be merged. Some assume that the models contain no contradictions. A totality of merge offers the benefit that we can perform exploratory merges and test whether the results are sensible, rather than sanitizing the models first.

6 Conclusions

In this paper, we argued that distributed model management depends crucially on being able to put together models coming from different sources, the process we referred to as *merge*. We have suggested a framework for comparing different approaches to merging, proposed a number of merge-related operators and discussed their desired properties. We have also illustrated the framework on two examples.

The resulting framework leads us to a reconsideration of modeling semantics based on how well they support distributed modeling. The intended relationship between two partial models may be at odds with the usual semantics of the modeling formalism. In particular few modeling languages address the notion of missing information (partiality). Notable exceptions for state modeling are approaches based on Live Sequence Charts [6], MTSs [9, 22] and partial Kripke structures [13]).

The examples provided in this paper are *homogeneous* merges: the input models and output models are all of the same type. However, we believe our algebraic approach generalizes to heterogeneous merging, like the one advocated in [6].

Throughout the paper, we have argued for a more explicit specification of the relationships between models during distributed model development. In our future work, we intend to devote more attention to the question of how to elicit and represent such relationships between models.

Finally, we would urge the model management community to launch an effort to gather benchmarks for various aspects of the merge process. The benchmarks should include some description of the domain (what is being merged), and specific examples of models and their relationships. The gathered benchmarks will help move the field forward in many ways: (1) provide grounds for comparison between the different approaches to merging; (2) stress-test the tools; (3) help to drive the research forward by improving consensus on a shared research agenda [21].

References

[1] M. Barr and C. Wells. *"Category Theory for Computing Science"*. Les Publications CRM Montréal, Montreal, Canada, 3rd edition, 1999.

[2] P. Buneman, S. Davidson, and A. Kosky. Theoretical aspects of schema merging. In *Proceedings of the 3rd International Conference on Extending Database Technology*, pages 152–167, 1992.

[3] S. Easterbrook and M. Chechik. "A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints". In *Proceedings of International Conference on Software Engineering (ICSE'01)*, pages 411–420, 2001.

[4] A. Egyed and N. Medvidovic. "A Formal Approach to Heterogeneous Software Modeling". In *Proceedings of Formal Aspects of Software Engineering (FASE'00)*, pages 178–192, 2000.

[5] J. Goguen. "A Categorical Manifesto". *Mathematical Structures in Computer Science*, 1(1):49–67, 1991.

[6] D. Harel, H. Kugler, and A. Pnueli. "Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements.". In *Formal Methods in Software and Systems Modeling*, pages 309–324, 2005.

[7] S. Horwitz, J. Prins, and T. Reps. "Integrating Noninterfering Versions of Programs.". *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, 1989.

[8] S. C. Kleene. *Introduction to Metamathematics*. New York: Van Nostrand, 1952.

[9] K. Larsen and L. Xinxin. "Equation Solving Using Modal Transition Systems". In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 108–117, 1990.

[10] N. Liu, J. C. Grundy, and J. G. Hosking. "A Visual Language and Environment for Composing Web Services". In *Proceedings of Automated Software Engineering (ASE'05)*, pages 321–324, 2005.

[11] S. Melnik, E. Rahm, and P. Bernstein. "Rondo: a Programming Platform for Generic Model Management". In *SIGMOD Conference*, pages 193–204, 2003.

[12] R. Milner. *"Communication and Concurrency"*. Prentice-Hall, New York, 1989.

[13] S. Nejati and M. Chechik. "Let's Agree to Disagree". In *Proceedings of 20th IEEE International Conference on Automated Software Engineering (ASE'05)*, pages 287 – 290, 2005.

[14] S. Nejati, M. Sabetzadeh, M. Chechik, and S. Easterbrook. "Identifying and Representing Requirements Variability in Families of Reactive Software", 2006. Submitted for publication.

[15] B. Nuseibeh, S. Easterbrook, and A. Russo. "Making Inconsistency Respectable in Software Development". *The Journal of Systems and Software*, 58(2):171–180, 2001.

[16] R. Pottinger and P. Bernstein. "Merging Models Based on Given Correspondences". In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 862–873, 2003.

[17] E. Rahm and P. Bernstein. "A Survey of Approaches to Automatic Schema Matching". *The VLDB Journal*, 10(4):334–350, 2001.

[18] D. Richards. "Merging Individual Conceptual Models of Requirements". *Requirements Engineering*, 8(4):195–205, 2003.

[19] M. Sabetzadeh and S. Easterbrook. "An Algebraic Framework for Merging Incomplete and Inconsistent Views". In *13th IEEE International Requirements Engineering Conference*, September 2005.

[20] M. Sabetzadeh and S. Easterbrook. "Traceability in Viewpoint Merging: A Model Management Perspective". In *3rd International Workshop on Traceability in Emerging Forms of Software Engineering*, November 2005.

[21] S. E. Sim, S. Easterbrook, and R. C. Holt. "Using Benchmarking to Advance Research: a Challenge to Software Engineering". In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 74–83, 2003.

[22] S. Uchitel and M. Chechik. "Merging Partial Behavioural Models". In *Proceedings of 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 43–52, November 2004.

[23] Z. Xing and E. Stroulia. "UMLDiff: An Algorithm for Object-Oriented Design Differencing". In *Proceedings of 20th IEEE International Conference on Automated Software Engineering (ASE'05)*, pages 54–65, 2005.