Mirador: a Synthesis of Model Matching Strategies

Stephen C. Barrett
Concordia University
Montreal, QC, Canada
ste_barr@
encs.concordia.ca

Greg Butler
Concordia University
Montreal, QC, Canada
gregb@
encs.concordia.ca

Patrice Chalin
Concordia University
Montreal, QC, Canada
chalin@
encs.concordia.ca

ABSTRACT

Mirador is a model merging tool that supports multiple model comparison strategies for the purpose of matching model elements. Capable of running either standalone, or as a Fujaba plug-in, Mirador leverages the CoObRA software versioning package to obtain model change information.

The bringing together of various comparison strategies allows Mirador to solicit measures of element similarity from one or more strategies, as appropriate for a given matching context. As an addition to this strategy mix we suggest one based on model evolution, and illustrate its potential for use with some simple examples.

Mirador performs operation-based merging, premised on the notion of a plane of change operations, which we have extended into the third dimension to enable the detection of cross-matching strategy conflicts. We also propose breaking this monolithic change plane up into a series of local change planes to facilitate effective, conflict free merging.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering—*model merging tool*

Keywords

MDE, Mirador, model merging, operation-based merging, local merging, change plane, element matching strategy

1. INTRODUCTION

To a great extent the success of model-driven engineering (MDE) depends on the exploitation of automation [22], and nowhere is the necessity for automation more acute than in the case of concurrent development: modeling in parallel inevitably leads to model divergence and conflicts. Consequently, there is a great need for the research and development of tools which provide for the synchronization and merging of models [19].

Our own conclusion is that state-of-the-art tools fail to deliver an acceptable degree of automation [2]. Much of this state can be attributed to an inability to properly compare models. That is to say, establishing an accurate correspondence between elements of the models to be merged is critical. Beyond that, merging itself relies on model comparison to assess what differences need to be reconciled. Though many ideas for comparing models have been put forth, their focus, and hence the nature of their results differ widely.

This is the conundrum that confronted us as we set out on developing a model merging tool. Rather than try to come up with a definitive answer, or decide on the "best" approach, we decided to let Mirador become a platform from which various solutions could be exercised—separately, *or* in conjunction with other solutions. In this way the tool could grow to offer a synthesis of model comparison techniques as applied to the problem of matching model elements, in addition to providing a testing ground for our ideas on operation-based model merging.

A prototype has since been implemented as a plug-in for the Fujaba Tool Suite [8], which first came to our attention while researching use case modeling [1], and merging [3]. By happenstance, Fujaba uses a software configuration framework called CoObRA [21] to store its models as sequences of change operations—a great advantage for merge tools requiring access to a model's change history.

Besides the tool itself, this paper also explores the use of model transformations composed of primitive change operations as the basis for merging. The operations, which are thought of as residing in a plane, are woven together into merge transformations. To this abstraction we add a third dimension to reflect the need for multiple element matching strategies in the merge process, and show how comparison conflicts across this *strategy dimension* are manifested in the representation. A comparison strategy based on model change history is also suggested.

A refinement proposes breaking the monolithic *change plane* up into several independent local change planes. This is with an eye towards making the merge process more computationally manageable, and certain types of conflicts more discernible. An example is offered to illustrate how such an approach could work in practice.

The motivation for model merging, approaches to it, and the notion of treating it as a weave of subtransformations is explored next. Section 3 puts the weaves onto a plane of change operations where inconsistencies and conflicts can be detected and resolved. To accommodate multiple matching strategies, and help visualize comparison conflicts, Section 4 adds a third dimension to the change plane. In Section 5 we introduce our model merging tool Mirador, highlighting its

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IWMCP '10, July 1, 2010 Malaga, Spain
Copyright 2010 ACM 978-1-60558-960-2 ...$10.00.

element similarity measurement and matching functionality. Section 6 refines the change plane concept to use local merging. Related work is covered in Section 7. The conclusion and future work will be found in the last section.

2. BACKGROUND

Model merging is at the confluence of three broad areas of software development which drive the need for it, shape its issues and concerns, and influence how it is carried out: *model-driven engineering* (MDE) has been the impetus behind models moving to the center of the development process; collaborative environments fostered by *cooperative engineering* ventures are dependent on being able to synchronize and merge their artifacts; and many of the challenges facing model merging are encompassed by the closely related topics of *data replication*, *software versioning*, and *model comparison* and *differencing* techniques.

2.1 Cooperative Engineering

The goal of a software engineering group is, in its simplest terms, to move a software system from some initial state to some more complete state. In a cooperative endeavor, movement is effected by multiple developers working in parallel on portions of the same source code, which must eventually be brought back into agreement. Under MDE, models, not code are the focus, making the group goal more precisely one of moving incomplete or abstract models to more complete or concrete models.

Thus it is models that diverge as work proceeds. Model merging tools then, attempt to bring two or more **replicas** (i.e., initially identical copies) of some common ancestor model into agreement. Typically merging:

- requires a great deal of knowledgeable human input;
- quickly overwhelms the user with information;
- forces decisions amenable to automation on to the user;
- offers inadequate choices for conflict resolution [27];
- and, does cross-model element pairing at only a coarse level of granularity [13].

2.2 Merge Types and Approaches

The merge of two **replica** models M_L and M_R that have evolved from a common ancestor M_A, and are eventually combined into a final merged model M_F is usually depicted as in Figure 1. Since a **three-way merge** as it is known, can avoid inconsistencies by rearranging changes, as well as reference the common ancestor to resolve the so-called **add-delete problem**, merge tools make extensive use of it [16].

Approaches taken to merging fall into two broad categories: those that are predominantly state-based and rely on a static view of the replicas to be merged, and those that are operation-based and analyze a dynamic trace of model changes gathered during replica modification. Operation-based merging is 3-way by its nature, since there is no ambiguity over whether an element was added to one side, or deleted from the other. However, it does require tighter coupling with the modeling tool than the state-based approach.

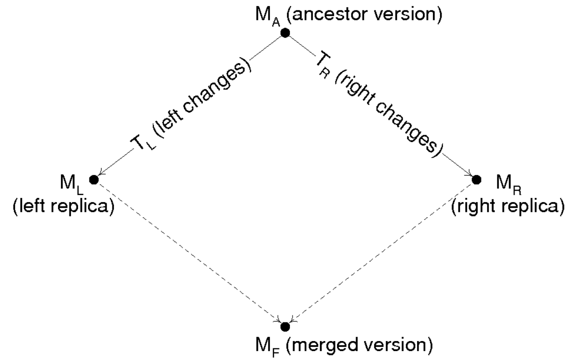

Figure 1: Merging two replicas derived from a common ancestor model.

2.3 The Merge as a Transformation

Each change made to a replica potentially transforms it into a new model. Model changes then, are atomic operations that can be treated as functions on models. Functional composition can be used to define an endogenous **transformation** T_e that takes as its input some initial model M_a conforming to some metamodel, and produces as its output a final model M_f conforming to the same metamodel:

$T_e M_a = (T_{e_n} \circ T_{e_{n-1}} \circ \ldots \circ T_{e_1}) M_a = M_f$

where T_{e_j} is a **subtransformation** of T_e. A merge then, can be expressed as a weave of the sub-, or primitive transformations laid out in a **transformation grid** [15].

3. THE CHANGE PLANE

In Figure 2 we orient the transformation grid to resemble the classic merge diamond. With model M_A situated at the origin, the axes represent transformations that produce new versions of it—one on the left, and one on the right—each being made up of subtransformations (directed segments). To traverse a path from the origin to another node is to execute the sequence of elementary change operations that lie on the path, transforming M_A in the process. Hence, every grid point hosts the set of models produced by all paths that reach that point. We term the resulting two-dimensional surface of model subtransformations and potential models, the **change plane**.

From a transformational point of view, to move forward

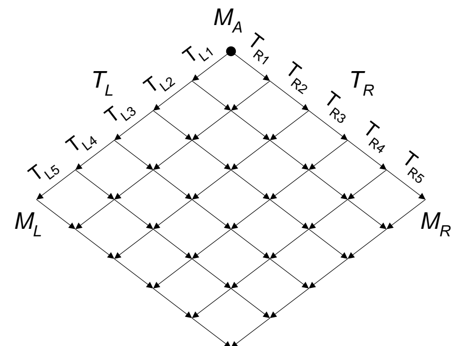

Figure 2: Model change operations projected onto the merge change plane.

(i.e., out from the origin) over an edge is to *accept* the change that the edge represents, while to avoid traversing an edge is to *reject* its change. The essence of merging is to blend the subtransformations of T_L and T_R into a single transformation incorporating as many changes from both sides as possible, which when applied to M_A yields a consistent merged model.

Absent any conflicts, transformations may be combined as wholes—tip-to-tail fashion as with vectors, otherwise they must be cobbled together piecemeal, subtransformation by subtransformation. This latter option is only truly possible in operation-based merging where the historical change-operations are available for perusal.

The merging of change operations can result in three kinds of errors. An **inconsistency** occurs if any operation in the path fails (e.g., referencing a nonexistent element). A syntactic **inconformity** results when an operation produces a model that is not well-formed (e.g., a nonunique ID). A semantic **conflict** is a contradictory pair of operations (e.g., naming the same element differently). Inconsistencies and inconformities can often be avoided, or reordered around. Conflicts are resolved by arbitrarily picking one change over the other, employing heuristics, or by appealing to the user.

3.1 Commutation and Conflict Resolution

Paths that respect the original ordering of their operations relative to a given change plane (that is, T_{L_i} comes before $T_{L_{i+1}}$, even if interleaved by T_{R_j}), *and* result in a consistent model are known as **weaves**. A path with an operation whose specification cannot be met will fail, and thus is not a weave: the path is prevented from reaching its target node due to an inconsistency. Weaves that terminate at the same node, but produce different outcomes are in conflict: a choice as to which weave to follow must be made, typically with knowledge a tool does not possess.

Noncommutative operations across merge transformations are at the root of conflicts, because a node's weaves differ only in the order of their subtransformations. If operations T_1 and T_2 commute globally, i.e., $T_1 \circ T_2 = T_2 \circ T_1$, then they will not conflict. Additionally, transformations that do not commute globally *may* commute locally when operating on mutually removed sections of a model, i.e., $(T_1 \circ T_2)M = (T_2 \circ T_1)M$, and consequently not conflict when transforming M [15]. Checking for operation commutation then is an appropriate way of determining if transformations are capable of being merged, and for isolating the causes of conflicts for purposes of operation deletion or modification. Likewise, inconsistency resolution can often take advantage of commutativity to find another path in which the circumstance that causes the inconsistency does not occur.

A node that hosts the same model for every one of its weaves is said to be **single-valued**. If any of the subtransformations are not at least locally commutative, then the node hosts more than one model and is said to be **multiple-valued**. The multiple-valued points that border the plane's single-valued nodes is called the **frontier set** [15]. It is the boundary at which operation-based merging can no longer automatically acquire conflict-free transformations. It is clear that the further the frontier set can be pushed out from the origin, the later merge decisions will have to be made, and the more context there will be to make them.

Consider a simple class model consisting of classes A and B. In part a of Figure 3 the ancestor model is modified by

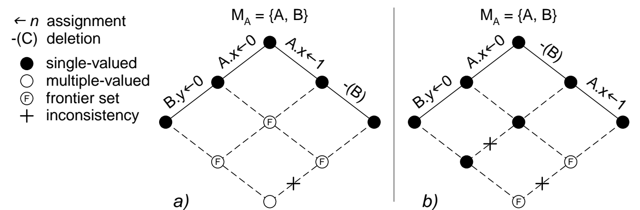

Figure 3: Using operation commutation to push back the frontier set.

the left transformation to initialize members x of A, and y of B to 0, while the right transformation sets $A.x$ to 1 before deleting B. All of the points along the axes are single-valued since there is only one path that reaches any one of them (disallowing backward travel). Because $A.x \leftarrow 0$ and $A.x \leftarrow 1$ do not commute, the two weaves that terminate at the center node produce different models, making the node multiple-valued. The rest of the non-axes points are multiple-valued for the same reason. Additionally, some of the paths going to the outermost node result in an inconsistency, because attempting to access B after it has been deleted violates a precondition of assignment, namely that the target exist. Overall, for this structuring of transformations, only those nodes that lie on the axes have no conflicts.

Swapping the position of the right hand subtransformations as is done in part b, eliminates some of the conflicts, turning two more nodes into single-valued points. The frontier set has thus been expanded slightly: Whereas the first structuring could only incorporate two changes that come from the same side (i.e., no merge), the second can incorporate one entire side and half of the other. This comes at the negligible expense of another inconsistency.

4. THE MATCHING DIMENSION

Before the change sequences of two models can be used for merging, each element in one model must be paired with its corresponding entity in the other model, if such an entity exists. Establishing this correspondence requires accurate model comparison capabilities, and is the key to successful model merging and conflict identification [23]. Indeed, once the two schemata are properly matched, model merging can proceed along essentially mathematical lines [18].

Alas, element matching is not without difficulties. Imagine that two developers working with the simple model in the topmost row of Figure 4 each add a new class B to their respective replicas (second row). The design intentions for Class B may, or may not be the same; without further information it is impossible to decide, and yet tools are forced to make just such a decision. For example, in this

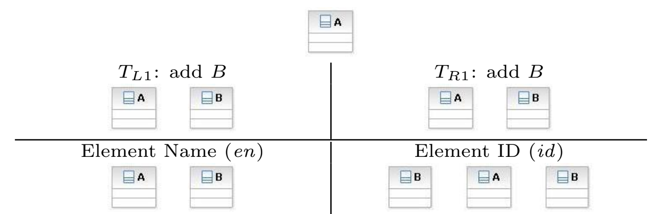

Figure 4: "Add same class" matching example.

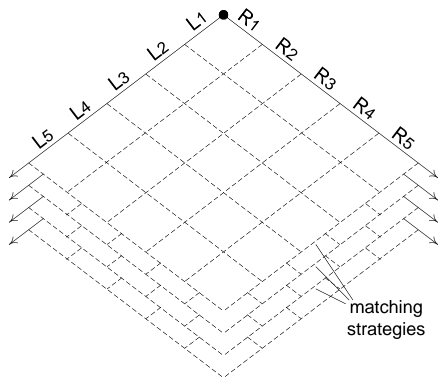

Figure 5: Matching strategies spread over a family of change planes.

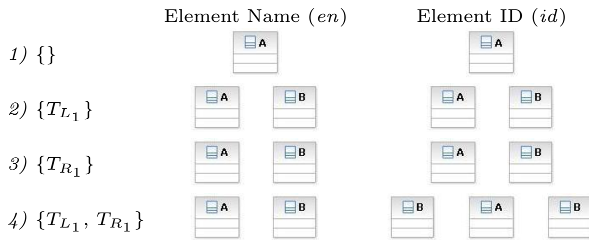

Figure 6: Session permutations for "add same class".

case IBM Rational Software Architect (RSA) decides that the classes are different, and produces a model (lower right) that suffers from the **duplicate element problem**, while IBM Rational Rose (Rose) draws the opposite conclusion, and combines the new classes into one (lower left).

Here, two different matching strategy are contrasted: one based on element ID, and the other on element name. Unfortunately the user is usually stuck with whatever strategy their tool is built to exploit. Mirador makes the outcome of both of these strategies, as well as others, available for inspection in the form of similarity measures, which can then be used for automatic, or interactive element matching (see Section 5.2).

Conceptually, this feature adds a third dimension to the change plane, in effect creating a new change plane for each strategy that is employed. The family of parallel planes thus formed (Figure 5) defines a generalized 3-dimensional solution space of merged models. The models for a given merge session then, are those produced by the weaves of the chosen planes, generated in accordance with their matching strategies.

4.1 Conflicts across Matching Strategies

Consider again the example of Figure 4 with both modelers adding a class B. At merge time, one, both, or neither of these changes may be accepted. The possibilities of merge session responses are laid out in the first column of Figure 6. The results of executing the transformations defined by each possibility appear in the columns to its right—one for each matching strategy.

Taken in isolation, the nodes of the two resulting change planes, are all single-valued: The models produced at every node are the same irrespective of the path traveled to reach the node. However, if we consider *crossing* from one plane to the other, the node reached by the fourth weave becomes multiple-valued! This can be seen in Figure 7, which places alongside each node, a miniature representation of the model produced by its weaves, adorned with the permutation number of Figure 6 that produced it.

Reasoning by analogy, we conclude that in the same way that multiple-valued nodes in two dimensions indicate conflicts over the plane of change operations, multiple-valued nodes in three dimensions signify conflicts across the space of change operations and matching strategies. The diagram makes clear that without the benefit of a full complement of model comparison strategies, and a means by which to analyze their results, model element matching will regularly fail to deliver the high quality schema mapping required for automated merging.

With the development of Mirador we have taken first steps towards putting a full assortment of model comparison algorithms, and element matching strategies under control of the developer. We present our attempt in the next section.

5. THE MIRADOR MODEL MERGER

A Java tool called Mirador is currently being developed at Concordia University to eventually support the operation-based merging of software models, but more immediately to act as a test platform for the ideas discussed in this paper. More prototype than tool at present, Mirador is slowly taking form as we continue our investigations into model merging. Still, it is concrete enough to lend some reality to its intended functionality, and to give a feel for how it could be used in a collaborative development effort.

The tool can be seen in Figure 8 running as a plug-in of the Fujaba (From Uml to Java And Back Again) Tool Suite. As there are no direct dependencies on the suite, Mirador is capable of being run standalone. A firmer dependency exists however, on the format of the Fujaba model files as maintained by the CoObRA (Concurrent Object Replication frAmework) versioning software. This constraint is not as severe as first appears since there are currently versions of Fujaba and CoObRA that run under Eclipse [9].

Mirador is implemented as a wizard dialog with three panels meant to be negotiated in sequence. The first panel parses the Fujaba models (i.e., CoObRA change logs) that are to be compared. These are dubbed the "left" and "right" models, though no significance is assigned to this distinction at the outset—the user has an opportunity to assign significance later.

It is also possible to read in a file containing any model element pairings that might have been established in a previous merge session. This file is created by the second panel,

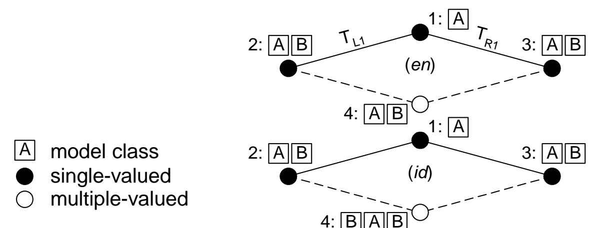

Figure 7: Cross-plane matching strategy conflict for "add same class" example.

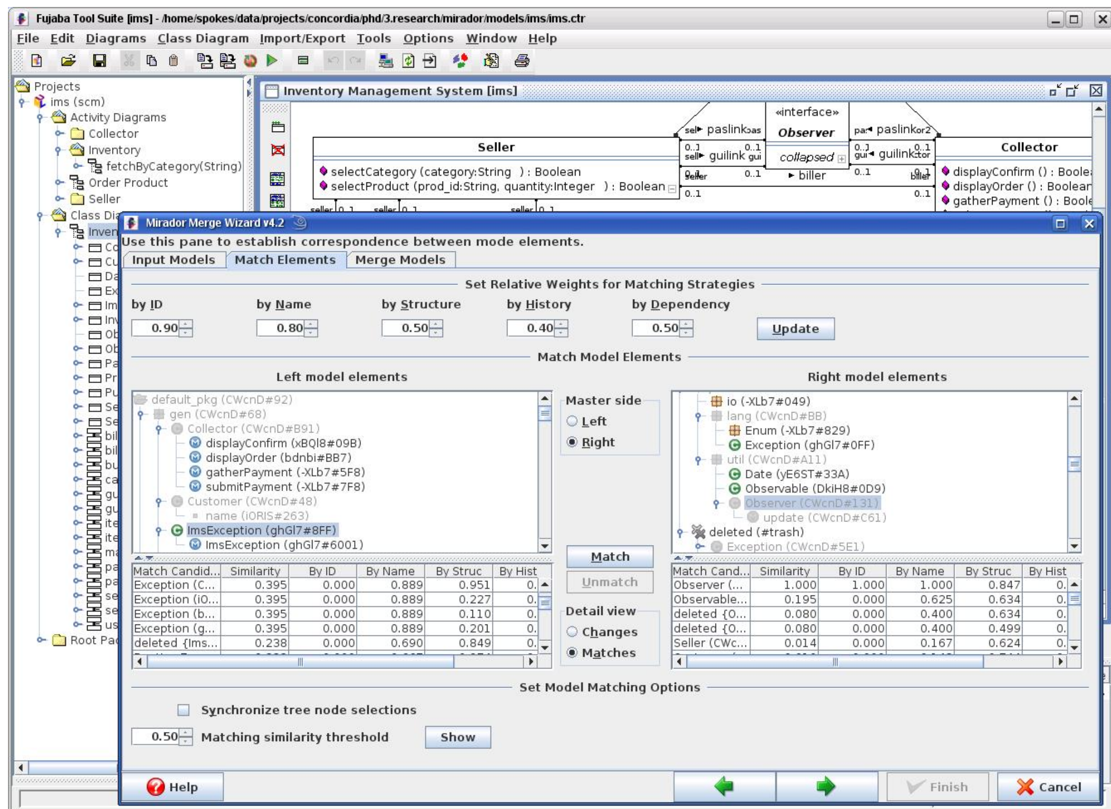

Figure 8: Mirador as a Fujaba plug-in, displaying similarity measures in the Element Matching Panel.

which is concerned with matching the elements of one model with those of the other. The matching may be accomplished automatically or interactively, or with a mixture of both. It is the most developed part of the tool, and the most relevant to the subject at hand. As such, it will be described in some detail over the next two subsections.

The last panel is where the actual model merging takes place. The work is carried out by weaving together the subtransformations of the merge change plane, as put forth in Section 3. A refinement to this method that we are considering implementing, after more analysis has been done, will be discussed in Section 6.

5.1 Element Matching Panel

Figures 8 and 9 contain screenshots of the Mirador Element Matching Panel. The GUI shows the left and right replicas of some common ancestor model as trees, which emphasizes the ownership and containment aspects of the models. Each tree node displays the name and ID of the model element it holds, along with an icon that reveals its kind. Hovering over a node brings up more information. At this stage of development, not only does the interface have the virtue of simplicity, it also has the advantage of not conflating changes to the model with changes to its view. An all too common state of affairs that can lead the unwary modeler into several different traps [2].

Details of the selected tree element can be seen in the table located directly beneath its tree. Exactly *what* details, is controlled by the radio buttons located between the two tables. In Figure 9 it is the CoObRA change records (less many irrelevant ones) that were generated over the course of the modeling session, listed in their order of occurrence. The lists are very fine-grained and rather low level, but amenable to consolidation.

The detail tables of Figure 8 list all the possible matches for the selected tree elements. That is to say, all the elements of the same type that are to be found in the other model, ranked by similarity (see Section 5.2). It should be noted that deleted elements are also eligible for matching. Clicking on a table candidate will highlight the referenced element. Disabled tree elements (those that are grayed out) have already been matched, and are not available for pairing, though they still show in the details section. If the **Synchronize tree node selections** box is checked, selecting a paired element will highlight its partner. A pair can be broken apart by pressing the **Unmatch** button. Likewise, available elements may be paired by pressing **Match**.

5.2 Match Candidate Similarity

For each match candidate listed in a detail table, values of how similar it is to the selected element as measured by several comparison techniques are also reported. Additionally, a weighted distance function that uses these values to compute an overall similarity score is given. The means for setting the weights of this function may be seen running across the top of the Element Matching Panel, where there is an edit box for each model comparison strategy. The **Update** button will force a recalculation of the similarity scores with any newly set weights. Putting a strategy's weight to zero removes it from consideration in the calculation.

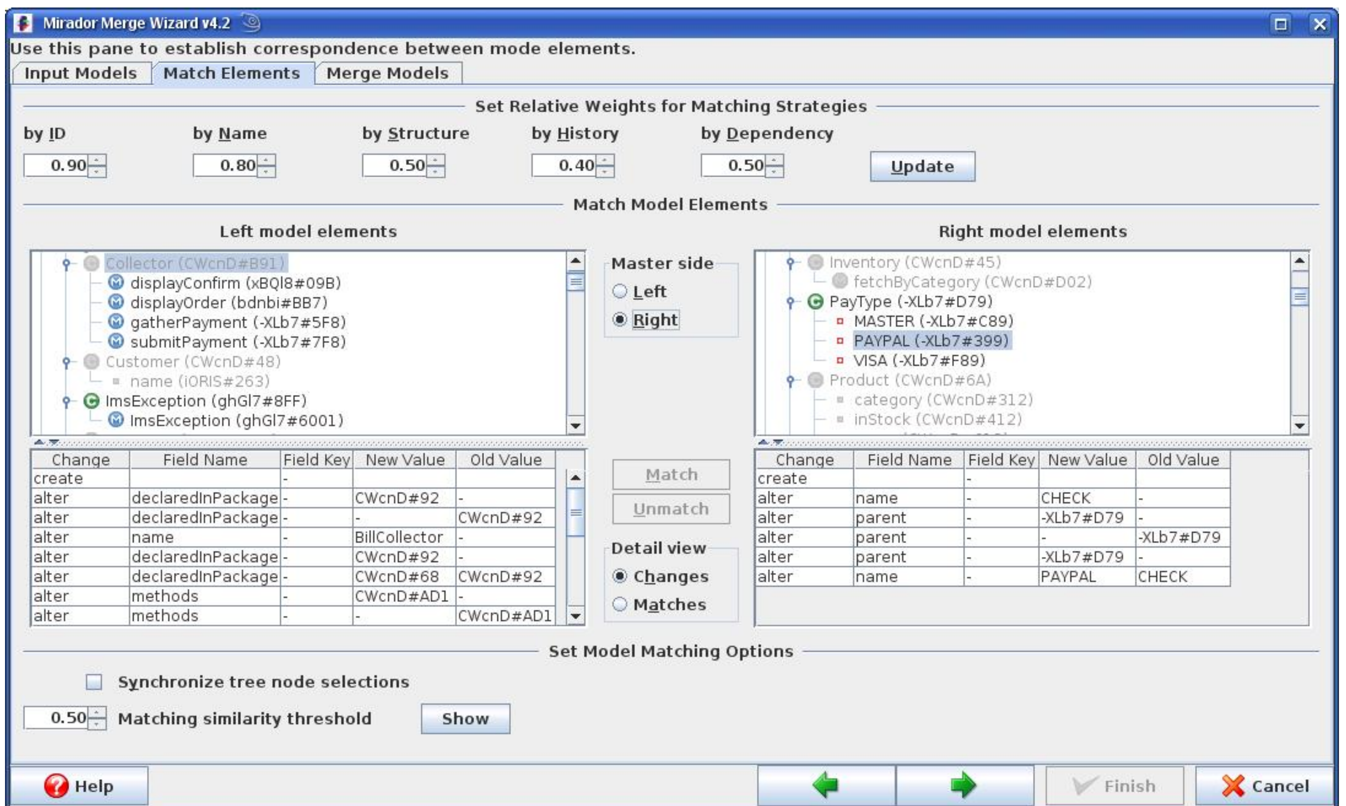

Figure 9: CoObRA change details as executed on the selected model elements.

The "by History" measure that can be seen in the figures, though only a placeholder at this time, is worth commenting on. The availability of history brings context and semantics with it, which model comparison could profit from in the same way that merging does. With this thought, we envision a time-based comparison technique that leverages the model change logs in order to seek out historical clues and insights for the purpose of element matching.

Once similarities have been calculated, elements can be automatically matched. The **Matching similarity threshold** edit box specifies the minimum similarity score, and the **Show** button may be used to highlight what elements satisfy the given threshold. Any elements not already paired when the user advances to the next panel will be automatically matched with the highest ranked candidate that is at or above the set threshold (assuming there is no other pairing with candidate that has a higher score).

The **Master side** radio buttons also come into play when moving from the Element Matching Panel to the Model Merge Panel. They specify which of the replicas is to drive the automatic facets of matching and merging. Its only purpose is to remove possible ambiguities. For instance, in the rare case when the similarity measures between two matchable elements are not symmetrical, the value chosen for ranking the match is dictated by the setting of this control.

6. LOCALIZED MERGING

The merge of two versions of some ancestor model can be effected by executing a transformation against the ancestor. On the change plane (Section 3), this merge transformation is seen to be a weave of subtransformations drawn from the changes made to the ancestor's replicas. Rather than construct one grand weave on a single change plane, we build many smaller weaves on their own change planes. The final merge is thus extracted from a series of individual change planes, each contributing their own *local* merge.

This refinement reduces complexity, and by taking advantage of local commutativity (Section 3.1), enables faster detection of merge conflicts. This in turn allows the troubled areas to be more quickly sent to the edges of the change plane, making for a rapid retreat of the frontier set. Mirador performs local merges on an element-by-element basis. As an example, consider the parallel changes made to a model of one package (ID #0) containing one class (ID #1), as shown here:

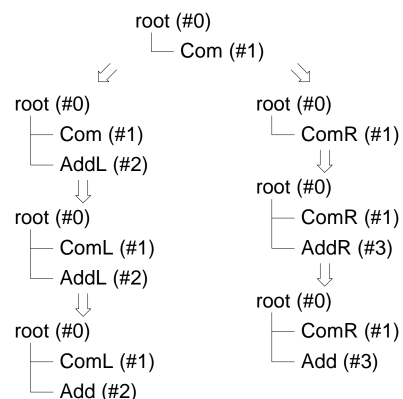

Table 1: Change transactions for left replica.

Tx#	ID	Change	Fld Name	New Val	Old Val
L1.1		create			
L1.2		alter	decInPkg	#0	
L1.3	#1	alter	decInPkg	-	#0
L1.4		alter	name	Com	-
L1.5		alter	decInPkg	#0	-
L2.1		create			
L2.2		alter	decInPkg	#0	-
L2.3	#2	alter	decInPkg	-	#0
L2.4		alter	name	AddL	-
L2.5		alter	decInPkg	#0	-
L3.1		alter	decInPkg	-	#0
L3.2	#1	alter	name	ComL	Com
L3.3		alter	decInPkg	#0	-
L4.1		alter	decInPkg	-	#0
L4.2	#2	alter	name	Add	AddL
L4.3		alter	decInPkg	#0	-

Table 2: Change transactions for right replica.

Tx#	ID	Change	Fld Name	New Val	Old Val
R1.1		create			
R1.2		alter	decInPkg	#0	-
R1.3	#1	alter	decInPkg	-	#0
R1.4		alter	name	Com	-
R1.5		alter	decInPkg	#0	-
R2.1		alter	decInPkg	-	#0
R2.2	#1	alter	name	ComR	Com
R2.3		alter	decInPkg	#0	-
R3.1		create			
R3.2		alter	decInPkg	#0	-
R3.3	#3	alter	decInPkg	-	#0
R3.4		alter	name	AddR	-
R3.5		alter	decInPkg	#0	-
R4.1		alter	decInPkg	-	#0
R4.2	#3	alter	name	Add	AddR
R4.3		alter	decInPkg	#0	-

The left and right records of the change operations (edited for brevity) that produced this model version trace have been taken from the Element Matching Panel and laid out in Tables 1 and 2. The changes have been grouped by CoObRA transactions, with subtransaction identifiers provided in the first column. Within each group, the original order of change execution has been maintained. The ID of the model element affected by a particular transaction is given in the tables' second column.

The first transaction is responsible for creating class *Com* in package *root*. It appears on both sides, because it is from the common model. Continuing on the left side: the second transaction, $L2$, creates class *AddL*, $L3$ renames *Com* to *ComL*, and $L4$ renames *AddL* to *Add*. Meanwhile, on the right: transaction $R2$ renames *Com* to *ComR*, $R3$ adds class *AddR*, and the last transaction changes its name to *Add*. When created, a class is declared in the default package ("decInPkg #0"). Upon being named it may move to another package (i.e., if a path is specified), hence the decInPkg reassignments, but in this example all remain in *root*.

If configured to use the "by ID" and "by Name" strategies with a threshold of 0.5, Mirador will match the elements with these measures: left class #1 to right class #1 (id=1.0, name=0.6, all=0.7), and left class #2 to right class #3 (id=0.0, name=1.0, all=0.5). The classes with identical IDs are matched, but with an overall similarity of less than 1. This reflects the fact that an ID *can* become decoupled from its original intent, say, if rather than deleting an unneeded entity, a developer co-opts it for another role. Our "by history" strategy (Section 5.2) could uncover such a role switch. It could be useful in a negative sense too: The pairing of element #2 with #3 results from a shared name. A search through change history could reveal whether this is a meaningful, or largely accidental fact.

6.1 Element-by-Element Merging

Accepting the suggested matchings, ignoring the common transaction, and proceeding element-by-element through the models produces a 3×3 change plane for the first pairing, and an 8×8 plane for the second. In the first case, suitable rearrangement of the change operations pushes, in Figure 10, the conflict between $L3.2$ and $R2.2$ as far away from the origin as possible.

In the larger plane, a similar conflict between $L2.4$ and $R3.4$ cannot be dealt with in the same manner. We are prevented from reordering these changes as we might desire by the existence of a noncommuting operation on both sides: $L4.2$ and $R4.2$, respectively. Fortunately in *this* situation, these later operations have the effect of overwriting the earlier ones, making the first namings superfluous—an outcome that is made more apparent by working with smaller merges, and which the Mirador operation pre-processor would be able to remove.

The plane contains another, more fundamental conflict: the create operations themselves, $L2.1$ and $R3.1$. Acceptance of an element matching implies that the paired elements are the *same* entity. But throughout the modeling phase, they have been considered to be entirely different entities—elements on the left interacting with the class whose ID is #2, and those on the right with the class having an ID of #3. Furthermore, neither class is even known in the other's model.

To resolve this conflict, two things need to happen. First, a choice as to which ID to retain going forward must be made—an ambiguity easily cleared up by the **Master side** radio buttons on the Element Matching Panel. And second, there must be a **dependency assumption** by the chosen element. That is, all elements in the opposite model that are dependent on the rejected element, must now switch there dependency to the chosen element. For CoObRA change records this involves simply replacing all occurrences of the old ID with the new one.

With all conflicts at the model element level resolved, the final step is to string together the local merges into an overall transformation. This sequence will include all the local transformations obtained through element-by-element merg-

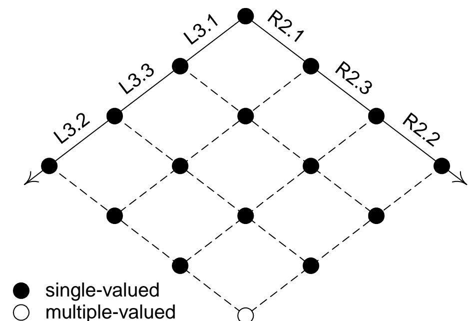

Figure 10: Local change plane for element #1.

ing, as well as any left behind subtransformations of the unpaired model elements. To avoid dependency problems when piecing together the final merge, it is best to follow the original ordering of the models in terms of element creation as closely as possible.

The resulting merge transformation will be conflict free, but may introduce inconsistencies when applied against the base ancestor model. Many tools return merged model that suffer from this issue: most notoriously, the so-called **duplicate element problem**. To guard against such possibilities, we are in the process of changing Mirador to use the Epsilon Validation Language [14] to post-process its result.

7. RELATED WORK

Model merging has yet to be adequately defined; Brunet, et al. [4] attempt to put it on solid footing by treating a merge as an algebraic operator over models and model relationships. In [5] they describe a fully automated merge of behavioral models based on their model transition system formalism. Somewhat unrealisticly, they assume that the replicas to be merged are fully compatible—conflicts, contradictions, and inconsistencies are not considered.

The approaches taken by tools to merging fall into two broad categories: those that are predominantly state-based, and rely on a static view of the replicas to be merged as exemplified by the Unison file synchronizer [17] and Rational Rose; and those like IceCube [10] that are operation-based, and analyze a dynamic trace of model changes gathered during replica modification.

Operation-based merging was advanced by Lippe and Oosterom [15] who described a transformation grid in which merges of object-oriented database changes are weaves of primitive grid transformations. They related operation commutation to conflict detection, and proposed three merge algorithms. We have adopted their transformation grid as our change plane, and extended it with a third dimension for matching strategies, and proposed using individual change planes on an element-by-element basis for local merging. Work done in the area of graph transformations (e.g., [7]), specifically with regards to critical pairs, and term and graph rewriting, touch on similar concepts. A recent resurgence of interest in operation-based merging, and the detection and resolution of conflicts is typified in the works of Koegel, et al. [11] and Schmidt, et al. [20].

Properly matching model elements is a problem of establishing identity. It is vital to successful merging, and approaches to it have been varied. Working on databases, Lippe and Oosterom did not have to concern themselves with entity matching. Others like Pottinger and Bernstein take it as a given [18]. And many tools such as RSA insist on unique element identifiers to do any useful work. More robust and flexible ideas are offered by Xing and Stroulia [27] who heuristically measure name and structural similarity for matching, or Kolovos [12] who furnishes a general comparison language. Mirador draws from these techniques and others, to offer an assortment of strategies for comparing and matching elements.

Treude, et al. [24] have created a system called SiDiff for performing similarity-based differencing of large models. In [26], Wenzel and Kelter couple their SiDiff framework with an underlying repository system to create a tool environment for tracing and visualizing model evolution. This is in a sense, the reverse of our proposed history-based comparison strategy: Whereas they use model comparison to arrive at a mapping that in turn is used to trace a model's evolution, we want to use a model's evolution to help build a mapping that serves for model comparison.

8. CONCLUSION AND FUTURE WORK

As a by-product of its primary mission to merge models, Mirador provides a means of assessing multiple model comparison strategies and combining their results within a single platform. Though still a work in progress it has already proven useful for visualizing model changes. In this paper we have given a quick tour of its user interface with regards to the element matching facility, and its use of various comparison techniques to obtain similarity measurements.

Mirador is operation-based, and uses the concept of a plane of change operations, or primitive transformations to perform its merging. This idea was extended into the third dimension in order to describe the effect different model matching strategies have on merging. It was then demonstrated how the abstraction captures the notion of matching strategy conflicts. It was also argued that breaking the change plane into a series of smaller planes could make merging more local, and hence more effective. Also sketched out was the idea of history-based comparison, wherein element correspondence is better established by tracing the lineage of the elements being considered for matching over the course of their model's evolution.

At this point Mirador is still a prototype. We are currently in the process of rearchitecting it to support other forms of change information. An abstraction layer is being designed to stand between the change logs and the internal representations of the models to be merged. Once the Fujaba-CoObRA input module has been retrofitted to the new architecture we plan on implementing another to handle Eclipse-ChangeRecorder trace logs.

The new architecture will also bring about an update of Mirador's internal model representation to either directly use, or be able to generate models of the UML2, the EMF-based metamodel of the Unified Modeling Language [25]. The motivation behind this move is to enable the incorporation of the Epsilon family of model management languages [14] to facilitate the compare, merge, and validation aspects of Mirador's operation. Another motivation is to eventually take advantage of the presentation capabilities of EMF Compare [6], which would entail a full migration of Mirador to Eclipse.

The comparison strategies Mirador uses to match model elements is an area of focus as well. Besides completing implementations of the strategies we intend to adopt, we also want to begin work on our proposed history-based matching strategy. Also, to find if there is value in having the tool provide a timeline, or snapshots of model evolution to the user. The element trace visualization of [26] could be helpful in this regard.

Along this same line, the distance formula currently being used to compute overall similarities for match candidates is only a first attempt. Further investigation is needed to be able to form a meaningful interpretation of these measures, and to determine the best way to use them in combination.

Finally, extensibility is a concern. Adding the ability to load and use pre-existing implementations of comparison techniques or user-defined matching strategies is a priority. Just how to achieve this is not known at this time. One pos-

sibility is to rely on the mechanisms provided by both Fujaba and Eclipse that allow plug-ins to communicate with one and other without creating compilation dependencies. Whatever the solution, we feel that giving Mirador this flexibility will be a key to its acceptance, and to staying current.

9. REFERENCES

[1] S. Barrett, G. Butler, and P. Chalin. Techniques for use case modeling in Fujaba. In *ICCET '10: 2nd Internat. Conf. on Computer Engineering and Technology*, Apr 2010.

[2] S. Barrett, P. Chalin, and G. Butler. Model merging falls short of software engineering needs. In *MoDSE '08: Internat. Workshop on Model-Driven Software Evolution*, Apr 2008.

[3] S. Barrett, D. Sinnig, P. Chalin, and G. Butler. Merging of use case models: Semantic foundations. In *TASE '09: Internat. Sympos. on Theoretical Aspects of Software Engineering*, Jul 2009.

[4] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A manifesto for model merging. In *GaMMa '06: Internat. Workshop on Global Integrated Model Management*, pages 5–12, New York, NY, USA, May 2006. ACM.

[5] G. Brunet, M. Chechik, and S. Uchitel. Properties of behavioural model merging. In *FM '06: Internat. Conf. on Formal Methods*, Berlin, Germany, Aug 2006. Springer.

[6] E. Compare. Website of the EMF Compare subproject of the Eclipse EMFT project, May 2010. `http://www.eclipse.org/modeling/emft/?project=compare#compare`.

[7] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[8] Fujaba, Nov 2009. `http://www.fujaba.de/home.html`.

[9] Fujaba4Eclipse, Nov 2009. `http://www.fujaba.de/projects/fujaba4eclipse.html`.

[10] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In *PODC '01: 20th Annual ACM Sympos. on Principles of Distributed Computing*, pages 210–218, New York, NY, USA, Aug 2001. ACM.

[11] M. Koegel, J. Helming, and S. Seyboth. Operation-based conflict detection and resolution. In *CVSM '09: Workshop on Comparison and Versioning of Software Models*, pages 43–48, Los Alamitos, CA, USA, May 2009. IEEE Computer Society.

[12] D. S. Kolovos. Establishing correspondences between models with the epsilon comparison language. In *ECMDA-FA '09: 5th European Conf. on Model Driven Architecture - Foundations and Applications*, pages 146–157, Berlin, Germany, Jun 2009. Springer-Verlag.

[13] D. S. Kolovos, R. F. Paige, and F. A. Polack. Model comparison: a foundation for model composition and model transformation testing. In *GaMMa '06: Internat. Workshop on Global Integrated Model Management*, pages 13–20, New York, NY, USA, May 2006. ACM.

[14] D. S. Kolovos, L. Rose, R. F. Paige, and F. A. Polack. The Epsilon Book. Website of the Epsilon subproject of the Eclipse GMT project, 2010. `http://www.eclipse.org/gmt/epsilon/doc/book`.

[15] E. Lippe and N. van Oosterom. Operation-based merging. *ACM SIGSOFT Softw. Eng. Notes*, 17(5):78–87, Dec 1992.

[16] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. on Softw. Eng.*, 28(5):449–462, May 2002.

[17] B. C. Pierce and J. Vouillon. What's in Unison? A formal specification and reference implementation of a file synchronizer. Technical Report MS-CIS-03-36, University of Pennsylvania, Philadelphia, PA, USA, Feb 2004. `http://www.cis.upenn.edu/~bcpierce/papers/unisonspec.pdf`.

[18] R. A. Pottinger and P. A. Bernstein. Merging models based on given correspondences. In *VLDB '03: 29th Internat. Conf. on Very Large Data Bases*, pages 862–873. VLDB Endowment, Sep 2003.

[19] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Comput.*, 39(2):25–31, Feb 2006.

[20] M. Schmidt, S. Wenzel, T. Kehrer, and U. Kelter. History-based merging of models. In *CVSM '09: Internat. Workshop on Comparison and Versioning of Software Models*, pages 13–18, Los Alamitos, CA, USA, May 2009. IEEE Computer Society.

[21] C. Schneider, A. Zündorf, and J. Niere. CoObRA - a small step for development tools to collaborative environments. *IEEE Seminar Digests*, 2004(902):21–28, 2004.

[22] B. Selic. The pragmatics of model-driven development. *IEEE Softw.*, 20(5):19–25, Sep 2003.

[23] F. Taibi, F. M. Abbou, and M. J. Abbou. A matching approach for object-oriented formal specifications. *Journal of Object Technology*, 7(8):139–153, Dec 2008. `http://www.jot.fm/issues/issue_2008_11/article4`.

[24] C. Treude, S. Berlik, S. Wenzel, and U. Kelter. Difference computation of large models. In *ESEC-FSE '07: 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 295–304, New York, NY, USA, Sep 2007. ACM.

[25] UML2. Website of the UML2 subproject of the Eclipse MDT project, May 2010. `http://www.fujaba.de/home.html`.

[26] S. Wenzel and U. Kelter. Analyzing model evolution. In *ICSE '08: Internat. Conf. on Software Engineering*, pages 831–834, New York, NY, USA, May 2008. ACM.

[27] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *ASE '05: 20th IEEE/ACM Internat. Conf. on Automated Software Engineering*, pages 54–65, New York, NY, USA, Nov 2005. ACM.