BLENDING STATE DIFFERENCES AND CHANGE OPERATIONS FOR METAMODEL INDEPENDENT MERGING OF SOFTWARE MODELS

Stephen C. Barrett

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy
Concordia University
Montréal, Québec, Canada

April 2011

© Stephen C. Barrett, 2011

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By:	Mr. Stephen C. Barrett
Entitled:	Blending State Differences and Change Operations for Metamodel Independent Merging of Software Models

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Chair
Dr. Gerard J. Gouw

______________________________ External Examiner
Dr. Houari Sahraoui

______________________________ External to Program
Dr. Ferhat Khendek

______________________________ Examiner
Dr. Peter Grogono

______________________________ Examiner
Dr. Constantinos Constantinides

______________________________ Co-supervisor
Dr. Patrice Chalin

______________________________ Thesis Supervisor
Dr. Gregory Butler

Approved ______________________________
Chair of Department or Graduate Program Director

______________ 20 ______ ______________________________
Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Abstract

Blending State Differences and Change Operations for Metamodel Independent Merging of Software Models

Stephen C. Barrett, Ph.D.

Concordia University, 2011

A typical model merging session: requires a great deal of knowledgeable input; does not provide rapid feedback; quickly overwhelms the user with details; fails to properly match elements; performs minimal conflict detection; offers conflict resolution choices that are inadequate and without semantics; and exhibits counter-intuitive behavior.

Viewing model merging as a *process*, this research defines a *hybrid merge workflow* that blends the best of the main approaches to merging, expressing its phases as *algebraic operators* for performing transformations on model and relationship data types. Normalization and denormalization phases *decouple* models from their originating tool and metamodel. State-based phases capture model differences in the model itself, establish element correspondence using multiply *matching strategies*, and extract change operations. Operation-based phases then partition and order the changes prior to the detection and *automatic resolution* of conflicts.

The work has culminated in a *prototype* that validates the workflow, while realizing several novel model merging ideas, which are evaluated with simple and involved test cases. Combining the hybrid merge approach with the semantic expressiveness of *decision tables*—open to user modification—and an *interactive and batch mode* of operation allows the tool, named *Mirador*, to successfully address, to varying degrees, *all* of the previously cited shortcomings.

Dedication

It was in tracing the individual tracks of glaciers from the heights of the Patagonia Andes down to a single lagoon of cobalt blue that the ideas described herein began to crystallize. The grandeur experienced at the many *miradores* along the trail inspired this work, while the Spanish word for "vantage point" furnished a fitting name for its prototype.

Grand views, of course, no matter how inspirational, were not enough to see this dissertation through to completion. It was the many people who came to my aid when I was in need, and who gave of themselves without hesitation that sustained me throughout this endeavor. It is those individuals who I must now thank.

Foremost are my co-supervisors, Dr. Gregory Butler and Dr. Patrice Chalin, whose perfect blend of hands-off management and just-in-time guidance, coupled with their knowledge, advice, and patience, allowed me to roam the intellectual landscape without causing myself undue harm. I also offer my deep appreciation to the members of my advisory committee.

I am deeply grateful to my unofficial mentors: Dr. Homa Javahery who's own academic journey showed me what being a Ph.D. candidate truly meant; and Dr. Daniel Sinnig who, in a collaborative effort, demonstrated how to produce, in the words of one reviewer, "a beautifully written paper" in only seven days.

In the crunch of proofing, Peter Vranckx displayed a mastery of the English language inversely proportional to his understanding of computer software—a fortuitous combination. My thanks to Sofia for insisting we go trekking among glaciers, and for much more ("nöff, nöff"); to Nancy, who told me I would succeed; and to Xingmin, who supplied the motivation to continue in the most disheartening of moments.

My deepest gratitude goes to my family for their emotional support, continuous encouragement and unquestioning love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The dual conceptions of model-driven engineering (MDE) and model-driven development (MDD) are attempts to shift the emphasis in software construction away from languages in which code may be *written*, to models from which code may be *generated*. This is a logical continuation of the trend towards more abstraction—the chief weapon in combatting the complexity inherent in software development, something models are better suited for than code [Lennhamn, 2007]. Through this change in focus, MDE and MDD aspire to increase developer productivity, and consequently reduce product time to market [Sendall and Kozaczynski, 2003].

Cooperative engineering as well, by way of collaborative development, aims at increasing productivity. Inevitably, work proceeding in parallel on the same system leads to artifact divergence, and when brought back together, to conflicts, which can require a great deal of effort to resolve. For models specifically, it has long been recognized that their highly structured data present a special problem for merging, "Synchronization: This one word sums [up] the largest challenge to Model Driven Development" [Haskins, 1997]. And further, that tools are essential for reconciling models undergoing continual coevolution. Indeed, to a great extent, the success of model-driven development depends on the exploitation of automation [Selic, 2003]. Consequently, there is a great need for further research into and development of tools that perform the synchronization and merging of models [Schmidt, 2006].

1.1 Problem Statement

Ideally, model merging should be capable of automatically bringing together the models of a software system sharing the same viewtype into a single, coherent, and consistent model without incurring corruption or suffering information loss. However, empirical studies of the state of model merging demonstrate that this ideal has not yet been fully met. Typically, a model merging session:

- requires a great deal of knowledgable human input specific to the models being merged, and much effort to obtain meaningful results [Barrett et al., 2008];
- does not provide quick feedback, nor offer a mode suitable for script or batch processing [Bendix et al., 2010];
- can quickly overwhelm the user with non-relevant information that can lead to confusion [Barrett et al., 2008];
- fails to properly match elements across models, or only does so at a coarse level of granularity [Kolovos et al., 2006a], with no provision for making corrections;
- performs only minimal detection of merge conflicts [Pierce and Vouillon, 2004], [Pierce et al., 2004];
- offers inadequate choices for conflict resolution [Xing and Stroulia, 2005];
- cannot use semantics for conflict resolution [Cicchetti et al., 2008]; and
- exhibits counter-intuitive behavior [Barrett et al., 2008].

Merge tools vary in the degree to which they are susceptible to these shortcomings, and their outcomes are often very different from one another. At the very least, the range of tool responses would suggest that there is *no common set of expectations* to which tool vendors conform. Seen in a more pessimistic light, it suggests that the consistent delivery of experiences as portrayed above, by tools as a whole, may be taken as evidence that the *problem of model merging is as yet unsolved.*

This research takes the stance that merging can be more properly understood as a process than as a discrete activity. The *process* of merging actually encompasses a range of activities including model differencing, model comparison, operation recording or extraction, element or operation selection, and conflict detection and resolution, each of which comes with its own issues. This process view is used to frame an approach that addresses, to varying degrees, *all* of the points raised above.

1.2 Research Objectives

The overall goal of this research is to *increase the viability of software model merging* in its role as a critical component of model-driven engineering. Central to this goal is understanding, defining, and improving the merge process itself. Contrasting an imaginary ideal merge session with what is typical of today's tools leads directly to the following objectives:

a. *Express the merge process.* It is desirable to have a way to document what transpires during a merge as an aid for communication and understanding.

b. *Enhance model merging usability.* The amount of information requested from, *and* presented to users should be minimal. Moreover, some feedback concerning the merge outcome should be obtainable with *no* user input.

c. *Open model element matching.* How elements of the models are matched should be dictated by the models—not the tool—and be amenable to user suggestions.

d. *Decouple from the modeling tool.* The process should be capable of merging models produced by a variety of tools, irrespective of their metamodels.

e. *Configure the merge.* A merge should be configurable to the model types involved, the domain being modeled, and even the special needs of a project.

f. *Identify troubling merges.* Situations that signal poor merge results should be cataloged for use in creating and evaluating merge algorithms and tools.

g. *Improve merge results with patterns and semantics.* Patterns known to be troublesome should be handled properly, and semantics added to strengthen results.

h. *Extend merge knowledge.* A mechanism should exist for extending merge accuracy with the addition of information on troubling patterns, or specific domains.

Though accepting of its importance, the community is divided over how best to accomplish model merging. Where one camp maintains that an approach based on model state will be most effective, the other insists that correctness requires making use of change operations recorded over the course of a modeling session. Of course, both claims have their merits, and as with many dichotomies, the differences between the two approaches are less than they might at first appear. By declining to take sides, the thesis is arrived at:

Thesis Statement. At different phases in the merge process, *one approach is more suitable than the other.* A process that blends its use of the approaches as appropriate can thus take advantage of the best aspects of each to come closer to the ideal merge.

Scope of Work. The proposed hybrid process is concerned only with merging, which creates a new artifact from those being merged; not synchronization, which results in existing artifacts becoming identical if at the same level of abstraction, or equivalent if at different levels. Furthermore, no attempt is made to formally defined just what constitues a merged model, nor which of it properties should be preserved by the process.

To demonstrate the feasibility of a hybrid merge process, a model merging prototype has been implemented. Being a proof of concept, the tool has several limitations: 1) Though separate from Eclipse, the tool uses Ecore for representing models internally, necessarily restricting its input to class diagrams; 2) Decoupling interfaces have been created for only two modeling tools: Eclipse and Fujaba; 3) Three element matching evaluators have been coded; the rest are merely placeholders; 4) No means for ensuring that the rules used to drive merging are consistent and complete has been devised, though limited checking, via assertions, is available; 5) The graphical user interface for displaying the models and change conflicts is not fully developed; 6) The merging of diagram views has not been pursued; and 7) Performance was not a design consideration.

In principle, the proposed approach can be used to merge any kind of models, albeit without behavior preserving semantics. But because the implementation is limited to class diagrams, this work focuses exclusively on the merging of these.

1.3 Research Contributions

The primary contribution resulting from this research is a complete process for the merging of software models, irrespective of the tool used to create the models and any recorder used to capture changes made to them. The best of the two main approaches to merging have been blended to create a process that does not suffer from the inadequacies of either. The addition of abstracting front- and back-ends, a decision table mechanism, and a text-based, command line-driven mode allows us to:

1. Merge models independent of metamodels—even *across* metamodels.
2. Decouple operation-based models from their recorder of change operations.
3. Perform model comparisons and establish element correspondence with multiple matching strategies.
4. Automatically detect and resolve change conflicts, opening the way to incorporating model semantics into merge decisions.
5. Reduce human decision-making, and provide a mode of operation for rapid feedback or automated processing.

In addition to these procedural contributions, the research also makes the following conceptual and practical contributions:

6. Clarifies the definitions of *synchronization* and *merging*.
7. Provides a visual means for reasoning about merging, complete with the effects of model element matching strategies.
8. Presents a change operation partitioning and ordering scheme that observes change dependencies, eliminates change cycles, and preserves merge context.
9. Collects primitive merge cases for the purpose of evaluating merge tools.

The contributions enumerated above have culminated in a model merging tool named *Mirador*. Mirador integrates the related but usually disconnected activities of model differencing, model element comparison and matching, and model merging into a smoothly functioning whole capable of conducting software models through the entire merge process—from initial artifact divergence to final merged outcome.

1.4 Organization

The background material of Chapter 2 looks at the differences between the operations of synchronization and merging, and examines those aspects of the software development process that drive the need for them and influence how they work. The types of merging, the approaches to it, and the errors that can result are also covered. The chapter concludes by exploring the idea of treating model merging as a model transformation.

In Chapter 3, the view of software development as recurring cycles of merge sessions is discussed. Next, the manifestation of some of the problems associated with model merging is demonstrated in a selection of current MDE tools. The problems are then generalized and summarized, and recommendations for addressing them made.

This notion of the merge as a model transformation is taken up again in Chapter 4. In it, a three-dimensional space in which the merging of transformations can occur is built up from a *transformation grid* proposed by Lippe and van Oosterom [1992]. Along with their grid comes the notion of a conflict free region, bordered by nodes hosting two or more models that they term the *frontier set*. We separate these nodes, which signify conflicts, into two types. We also clarify the semantics of the predicate they use to populate the grid and detect conflicts.

Also presented, is a matrix for noting change dependencies and conflicts, and a scheme for partitioning the changes for ordering purposes [Lippe and van Oosterom, 1992]. We make the matrix suitable for tracking both same-side and cross-side changes, and show how to break the cycles that may occur in partitioning.

Chapter 5 develops the thesis statement. It describes a workflow for model merging that is a hybrid of the two main approaches to the problem. This hybrid workflow forms the core around which we build a proof of concept prototype, the architecture and design of which are described in Chapters 6 and 7, respectively. In Chapter 5, the eight phases of the workflow are depicted in a pipe and filter system, and described using algebraic operators that transform model and relationship data types. Each phase and the contribution it makes to model merging is briefly introduced here.

The workflow decouples the merge process from the models' originating tools with its initial input and final output filters. These normalization and denormalization actions are unique to the workflow. Subsection 6.3.1 details their architecture and 7.2.1

their design. A comparison phase then establishes correspondence between elements of the models to be merged. An added control interface allows the implementation to intervene in the matching process, something no other tool supports. Another novel aspect of the implementation is the use of multiple strategies to effect this matching. The filter's architecture is discussed in Subsection 6.3.2, and its design in Section 7.3. A differencing phase determines what model elements have changed and how. The architecture is given in Subsection 6.3.1. Unique to the workflow, the deltas are captured in models conforming to a difference metamodel, which is explained, along with the phase's design, in Subsection 7.2.2.

These phases constitute a state-based front-end. The next two phases, working in tandem, transition the workflow to operation-based activity. This is the most critical contribution of the hybrid workflow. The transition begins with the extraction of change operations from the difference models of the previous phase; subsections 6.3.3 and 7.4.2 expand upon this action. The extracted operations are ordered and checked for contradictions in the next phase, thereby effecting the merge, albeit with conflicts. Ordering is added for useability purposes, and is new: state-based tools make no use of change order, and operation-based models, being essentially change logs, are implicitly ordered. Details of the filter are given in Subsections 6.3.3 and 7.4.2.

The prototype drives this phase with decision tables, which bring semantics to the merge process, a valuable contribution in itself, since it is exceedingly rare in merge tools [Saito and Shapiro, 2002], but also, because the rules have been made changeable. The design of the decision table mechanism is covered in Subsection 7.4.3. The conflicts are resolved in the next phase, again using decision tables. Its architecture and design are found in Subsection 6.3.3 and Subsection 7.4.2, respectively. The last phase before workflow output, executes the change operations of the preceding phase against the base model, to produce the conflict-free, merged model (Subsection 6.3.3).

Validation of our contributions is done in Chapter 8; related work is given in Chapter 9; and the conclusion of Chapter 10 summarizes our contributions, points out its limitations, and makes suggestions for future work.

Chapter 2

Background

Model merging is at the confluence of three broad practices in software development which drive the need for it, shape its issues and concerns, and influence its realization. First, model-driven engineering has been the impetus behind models moving to the center of the development process; second, the collaborative environments fostered by cooperative engineering ventures create a demand for artifact synchronization and merging; and third, the closely related fields of data replication and configuration management have given hints as to how model merging might best be carried out.

Before discussing the significance each of these practices has for model merging, the nature of synchronization and merging—terms that are often used interchangeably, but which have distinct meanings—are explored, and working definitions for the two arrived at. Then models and model-driven engineering are examined in the context of cooperative engineering.

Much of the work that has been done on the managing of artifacts undergoing parallel changes has been conducted in the areas of data replication and configuration management. Although not fundamentally concerned with models, both have grappled with some of the same challenges that face model merging, and as such can provide insight into the problem, as well as a language for reasoning about it.

The remaining sections of this chapter focus on merging, which is introduced through the conveyance of a model merging session. The main approaches to merging are explained, merge types and errors defined, and lastly, the act of merging is looked at in the light of model transformations.

2.1 Synchronizing versus Merging

The need for reconciliation during software development stems primarily from parallel, or concurrent, development [Haskins, 1997]; the norm for modern software engineering processes. Reasons *not* related to concurrency include reuse, and the integration of constituent subsystems or modules. Neither of which actually changes the nature of the reconciliation, just the motivation for undertaking it: whenever the content of workspaces that have been changed independently are to be brought back together, they must be somehow reconciled.

Reconciliation is accomplished through the operation of either synchronization or merging. The literature can be frustratingly vague about the distinction between the two terms, often using them in ways that suggest they are interchangeable. They are regularly used to paint a picture of bringing different datasets or parts of them into agreement, or *alignment*—the word we adopt to describe the condition of synchronized or merged datasets.

When alignment pertains to resources of a physical storage device, such as file systems or personal digital assistants, the operation mentioned most often is synchronization. When addressing the logical entities contained within these resources, like files or programs, synchronization and merging seem to get equal billing. And when the context is the individual elements that make up such logical entities, e.g., pages and code blocks, merging seems to be the overwhelming favorite—especially if there is a possibility of some sort of *conflict* occurring between elements.

Buried within the concept of synchronization is the notion of wholeness—entire resources or entities are synchronized. Merging, on the other hand, embodies the assumption of parts—constituent elements, or logical partitions of the whole. What is considered the whole, and what the part, is a relative matter. For example, the synchronization of two directories would treat them as wholes and their files as the parts, while at a lower level, two files would be thought of as wholes and their contained records as parts. Additionally, synchronization can be either "horizontal" (i.e., between entities at the same level of abstraction, or models conforming to the same metamodel), or "vertical" (i.e., entities at different abstraction levels, or models conforming to different metamodels) Xiong et al. [2007]. This work concentrates on horizontal synchronization.

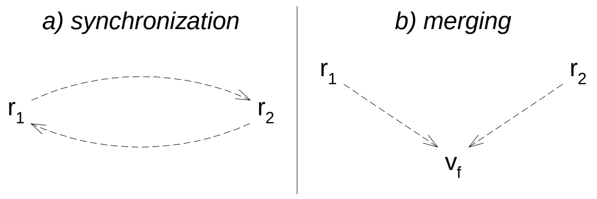


Figure 1: Synchronize operation versus merge operation

Both operations work with originals and *replicas*. When the scope is data, a replica is in all respects an exact copy of an original. Replicas can originate from a common source or from each other, or in the limiting case, from an empty or null dataset. If a replica is changed, it becomes a new *version* of the original. We often write *replica-version* as shorthand for expressing the idea of an initially exact copy of some original that has since been changed.

The major phases of synchronization are "update detection and reconciliation" [Pierce and Vouillon, 2004]. In general, the synchronization of storage devices ends with the devices having *identical* content. The number of unique resources does not grow, but existing resources may be duplicated and, in some cases, deleted. Synchronization by itself does not create any new versions. In contrast, merging takes a mix of originals and replicas and weaves them together to create a new version.

The line between the two operations tends to blur, because a merge is often launched *as a result of performing a synchronization operation* [Letkeman, 2005b], [Pierce et al., 2004]. Where synchronization can initiate a merge, the act of synchronizing replicas can appear to create a new version. Figure 1*a* shows an unspecified algorithm synchronizing two replicas by overwriting one with the other, or incorporating the independent updates made in one replica, into the other (via a merge) such that they are made identical. By contrast, the merge in Figure 1*b* clearly portrays the creation of a new version out of two replica-versions, which themselves remain unchanged. There is slight lose in generality in considering only two replicas in these operations, but not enough to be considered important.

The observations of this section lead to working definitions for synchronization and merging to be elaborated on in the later sections of this chapter:

> The process of bringing two or more work spaces or repositories of data, or replicated work items into alignment by making *all corresponding replicas*

identical will be called *synchronization*.

The act of aligning corresponding work item replicas by the creation of a new version, preserving *all non-conflicting replica updates* along with *updates that resolve the conflicts*, will be called *merging*.

2.2 Models and Model-Driven Engineering

Constructing, analyzing, and communicating in terms of models is at the heart of science and engineering: a characteristic that software engineering has in general, up to now, not shared with other engineering disciplines. This is something that MDE aims to change.

A *model* is an abstraction of some reality. A model or collection of models represents—to some level of accuracy—a system under study. It yields a set of statements, or propositions, about the system it represents. There are two possible ways of interpreting the proposition set: a model is said to be *correct* if all of its propositions about the system are true; conversely, a system is considered to be *valid* if it does not contradict any of its model's propositions [Seidewitz, 2003].

These last two statements reflect the ways in which models are used in practice, either as a description or as a specification. In science, a model is usually created in order to describe a system. The model is accepted as being correct if the statements the model makes correspond to observations made of the actual system. If this is the case, it becomes legitimate to analyze the system by reasoning about the model. To reason about a model, however, requires a theory, along with a means for mapping between model and system elements, and for relating the deductions made from the model back to the system. Newtonian physics, for example, provides the theory and mappings for modeling and reasoning about the solar system.

Engineering typically inverts the notion of a descriptive model, making it instead prescriptive, so as to specify a system. The as-built system is valid relative to its specification if it does not contradict any of the model's propositions. Once again, a theory and mappings are required to relate elements, specifications and constraints between model and system. An electrical schematic is an example of a model that uses a defined notation to map diagram symbols to system elements, and an electronic theory to prescribe how a circuit must behave.

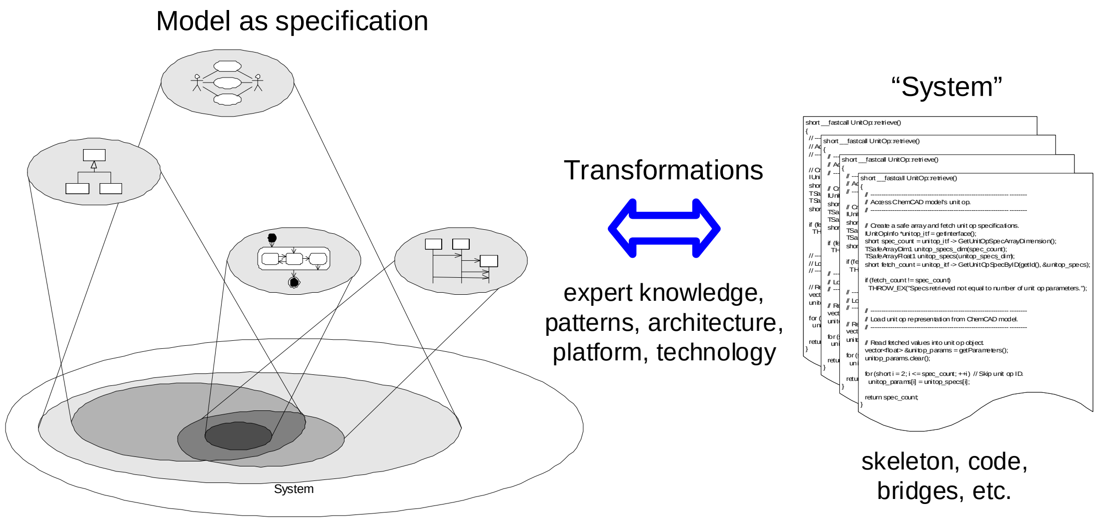
Models possess the following desirable qualities:

- *Abstraction.* A model hides irrelevant details and may only provide a limited *view* of some aspect of a system.
- *Understandability.* A model should have an intuitive form and be expressive.
- *Accuracy.* For the system under study, a model provides a true-to-life representation of its features of interest.
- *Predictiveness.* Non-obvious properties about the system under study can be drawn out of its model, either through experimentation or formal analysis.
- *Inexpensive.* To be worthwhile, a model must be considerably cheaper to construct and analyze than the system being modeled.

In its simplest terms, *model-driven engineering* is the notion that the model or models that specify a system can be readily *transformed* into that system. For software systems, this means transforming the model into the actual source code of the system [Mellor et al., 2003]. From the point of view of software model merging, it is of little importance whether the transformations operate under the rubric of MDE, MDD or MDA. In each case, "model" is the key word: each proposes describing a software system with formal models from which its code may be generated (Figure 2). As the nuances of the various technologies are inconsequential for our purposes, we will continue to refer to all model-driven methodologies under the generic heading of model-driven engineering, unless there is a reason to distinguish between them.

When modeling, the proposed solution to a real-world problem is represented by a model that captures the essence of the solution's structure and behavior. If software is the proposed solution, then its model can be used to automatically generate some portion of the actual solution, through one or more model transformations. While code itself might be considered to be a model of sorts, since it must be transformed by a compiler or interpreter in order to run, its level of abstraction is rather low.

A steady consequence of software research and development over the past five decades has been the continual raising of abstraction levels. All in an effort to help us program more in terms of our design intent, rather than in terms of whatever


Model as specification
"System"
Transformations
expert knowledge, patterns, architecture, platform, technology
skeleton, code, bridges, etc.
System

```
short __fastcall UnitOp::retrieve()
{
  // ----------------------------------------------------------------
  // Access ChemCAD model's unit op.
  // ----------------------------------------------------------------

  // Create a safe array and fetch unit op specifications.
  IUnitOpInfo *unitop_itf = getInterface();
  short spec_count = unitop_itf -> GetUnitOpSpecArrayDimension();
  TSafeArrayDim1 unitop_specs_dim(spec_count);
  TSafeArrayFloat1 unitop_specs(unitop_specs_dim);
  short fetch_count = unitop_itf -> GetUnitOpSpecByID(getId(), &unitop_specs);

  if (fetch_count != spec_count)
    THROW_EX("Specs retrieved not equal to number of unit op parameters.");


  // ----------------------------------------------------------------
  // Load unit op representation from ChemCAD model.
  // ----------------------------------------------------------------

  // Read fetched values into unit op object.
  vector<float> &unitop_params = getParameters();
  unitop_params.clear();

  for (short i = 2; i <= spec_count; ++i)  // Skip unit op ID.
    unitop_params[i] = unitop_specs[i];

  return spec_count;
}
```

Figure 2: Schematic of code generation and reverse engineering

computing technologies happen to be used. Unfortunately, the complexity of today's platforms—increasingly layered with middleware and enterprise applications—threatens to impede this historic trend. The great hope of MDE is to circumvent such impediments and continue the steady upward march in abstraction.

Not only does model-driven engineering seek to further raise the level of abstraction at which we do our work, it also proposes to push the last remnants of low-level constructs, still being fashioned in the solution space of the underlying computing environment, into the problem space of the system under design [Schmidt, 2006]. The goal being to rid our system descriptions, once and for all, of any "computer-oriented" concepts. In short, MDE promises to lift us above the complexity of technological detail by enabling us to precisely describe a system through its models.

2.3 Cooperative Engineering

Cooperation occurs when a group of people pursue the same goal for the group's benefit. In engineering groups, the outcome of cooperative work almost always involves the construction of some sort of system. *Cooperative engineering* attempts to support this constructive goal by defining a process and controlling the movement of work items through it. The most brute-force strategy is to serialize the work, allowing only one person at a time to change the state of the system. An obvious improvement is to divide the system into smaller pieces to allow two or more persons to make changes to different parts of the system in parallel.

The number of divisions a system can undergo is ultimately limited: either a point of diminishing returns is reached, or it becomes plainly impossible to go further. Also, because the parts of a system must, by definition, interact, they cannot exist in isolation. Any walls thrown up to ward off extraneous changes will at some point have to be breached in the name of getting work done—something any cooperative work process must anticipate and deal with. Provisions for synchronizing or merging work artifacts, then, is a requirement of any cooperative engineering endeavor.

When the cooperative venture is general software engineering, the goal becomes one of moving a software system from some initial state, say conception, to some new state, say deployed. When the venture is *model-driven* software engineering, the goal is refined to become one of moving incomplete, or abstract models to more complete, or more concrete models.

People making modifications to the same system in parallel will inevitably step on each other's work, requiring that the tainted artifacts eventually be brought back into alignment. Modeling, even more so than programming, needs sophisticated support for reconciling diverging replicas if it is to avoid corruption of its work items [Letkeman, 2005b]. Without viable merging, models will be unable to play their intended role in the development process, "[i]n particular, sophisticated support for merging model versions is urgently needed" [Westfechtel, 2010]. If merging takes too much effort, is questionable, or cannot be trusted, a cooperative MDE project will be in danger of descending into a code-centric one—the difficulty of keeping the models up-to-date outweighing the benefit of using them for specification.

2.3.1 Cooperative Copies and Workspaces

Most software development activities require exclusive access to at least some portion of the total system data, for some arbitrary amount of time. Thus, parallel software development needs a process that supports multiple replicas, so-called *cooperative copies* [Estublier and Garcia, 2005] of the software modules under development. How these cooperative copies are to be created and later reconciled is of primary concern for cooperative processes.

In some environments, such as Computer Supported Cooperative Work (CSCW) systems, concurrent changes may be propagated "instantaneously," thereby keeping all replicas continuously synchronized. This *blackboard metaphor* is not workable for software engineering environments, however. This is because most software changes are made in the context of a single task (e.g., new function, algorithm update, bug fix, etc.), and the series of partial modifications necessary to accomplish a given task are best made and tested independently of any other tasks. Interleaving partial modifications from all concurrent tasks would leave the system in a confusing and, in all likelihood, inconsistent state, preventing compilation and testing until after *all* work had been completed, clearly an unworkable situation.

What is needed in such circumstances is a separation between the originating work item source and its cooperative copies—a concept known as *workspace isolation* [Estublier and Garcia, 2005]. Workspaces allow development activities to proceed simultaneously and independently by hosting isolated software replicas over arbitrarily long periods of time. Contrary to the way CSCW systems synchronize continuously, the synchronization of workspaces is initiated explicitly on an as-needed basis. Synchronization usually involves complete and consistent units of work, which are defined by the intended purpose of the workspace, and the policies of the associated development process.

2.3.2 Process Policies and Correctness

While tools can create and manage private workspaces for cooperative engineering, until relatively recently, no tools provided for the coordination of developers working concurrently [Estublier, 2000], but this is slowly changing [Estublier et al., 2005]. In the meantime, workplace policies must be established to fill the gap. A *process policy* identifies between which workspaces synchronization is to take place and when. It is an organization's collection of process policies that clarifies just what "complete and consistent" means with regard to units of work, and the schedule by which these units are to be kept synchronized.

There is no adequate global criteria, such as serialization, for enforcing the correctness of concurrent software modifications. This makes reconciling software work items fundamentally an issue of merge control, which in the end must concern itself with two questions:

- *Data consistency.* Does the alignment of a pair of changed units of work yield consistent results?
- *Process.* Are modifications to a unit or units of work being made by the right person, on the right unit(s), at the right time?

Without any other criterion, merging then becomes a substitute for serialization, the only well-known consistency criteria. A rule of thumb for judging the consistency of a code merge is that the operation results in “a source file that compiles (syntactic merging) and exhibits the intended behavior of each of the merged changes (semantic merging)” [Estublier et al., 2005]. More mechanically, a correctness-preserving merge of two replica-versions derived from the same source will produce a single result, which is the same as the result that would be produced by applying the change operations in the proper sequence to the originating source. While this definition certainly has intuitive appeal, it leaves unanswered, “What is the ‘proper’ sequence?” Moreover, it is essentially a restatement of the difference between the state-based and operation-based approaches to merging, to be covered in Section 2.7.

How successful a merge can be depends on the nature of its data: for sets, a merge function exists that will always produce the correct result; for lists, only approximate merge functions exist, but they can often produce the right result; and for complex data structures, only specialized algorithms that take semantics into account can perform correct, or even meaningful, merges.

In the final analysis, Estublier and Garcia [2005] note that cooperative engineering is a balance between: 1) maintaining isolation until all work is completed, and 2) merging as often as possible. It is the policies of the development process, and *not* the workings of a tool, that should define and control that balance. They also recommend that policymakers keep the following rules of thumb in mind when trying to achieve this balance:

- Permit concurrent activity on only unrelated units of work, to reduce the probability of conflict.
- Merge as often as possible, thereby reducing both the number and range of conflicts, and making them easier to resolve.

- Give the merge task to the person who has the most intimate knowledge of the units to be merged, and thus the person most likely to do the job properly.

Though all good recommendations, the practicality of the first point is somewhat questionable for the early stages of most phases of any software project, when changes are occurring at their most rapid, and “units” may exist only fleetingly.

2.4 Data Replication and Synchronization

The act of *data replication* propagates the changes made to an item, using synchronization (with or without merging) to update all out-of-date replicas of the item to make them again identical. With replication, the label of “original” loses its meaning—being identical, replicas cannot be distinguished from one another, or from their originating source. To retain the notion of an original, one of the replicas must be held constant, while the others are allowed to change, as is the case with merging.

How a process interacts with its replicas implies a relationship that can be used to make a stronger distinction between synchronization and merging: data replication, as its name suggests, produces identical replicas, *via synchronization*; while configuration management, to be discussed in the next section, produces a string of new originals, or historic versions, *via merging.*

2.4.1 Traditional and Optimistic Replication

Replication maintains replicas of critical data at multiple, potentially connected computers or sites while allowing for access to any one of the replicas. Such systems face a tradeoff between availability and consistency. The increasing appearance of distributed data sharing systems, accompanied by issues of time-zone differences and network latencies, serves to complicate this balance.

Traditional data replication is pessimistic in nature. It offers the illusion of having a single, highly available, up-to-date copy of data. Pessimistic algorithms must rely on synchronous replica coordination to maintain this illusion. The basic concept is denial of access to data unless the data are provably up-to-date (Figure 3). The algorithms tend to update all replicas at once, possibly blocking read requests while

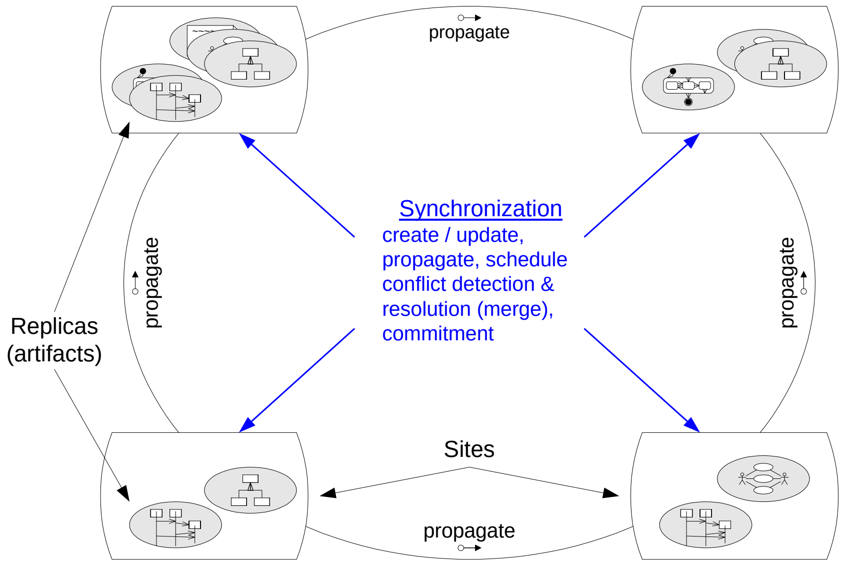


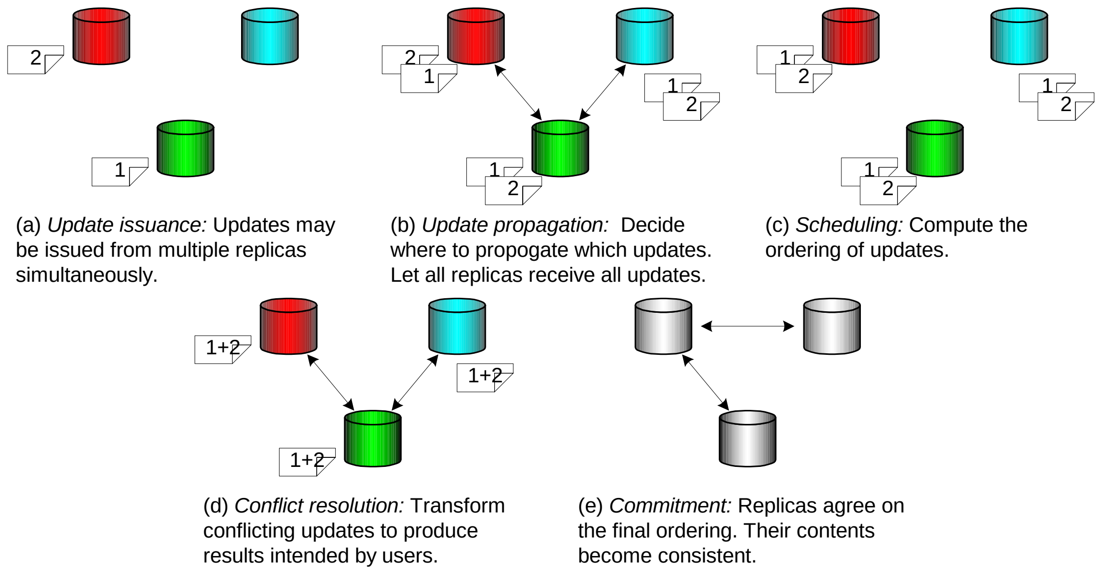
Figure 3: Schematic of data replication with synchronization capability

an update is underway. Thus, pessimistic systems keep replicas strictly consistent at the cost of sometimes making them inaccessible.

While suitable for many applications, the pessimistic "blackboard" approach to data sharing is not sufficient for cooperative engineering. We want to be able to work concurrently with copies of the same data. It is preferable to allow for concurrent updates in relative isolation, repairing the conflicts after they happen, than to lock data during editing. A key technology for enabling uncoordinated updates to data is optimistic data replication [Saito and Shapiro, 2002]. It allows for the divergence of replica content in the short term, only to reconcile the updated content in the long term, thus supplying the flexibility necessary for effective concurrent work.

Divergent replicas carrying and presenting inconsistent data, which must be fixed after the fact, is optimistic in the sense that conflicts are thought to be relatively rare, necessitating only the occasional repair. Even if this is not always the case, some human activities—among them cooperative engineering—inherently demand optimistic data sharing [Saito and Shapiro, 2002].

An optimistic approach guarantees any-time access at the expense of letting replica content diverge temporarily. The overall goals of an optimistic replication system can be stated as providing continual access to "sufficiently fresh" data, while at the same time minimizing user confusion that might arise due to replica divergence. Optimistic



2
1
(a) *Update issuance:* Updates may be issued from multiple replicas simultaneously.
2
1
1
2
1
2
(b) *Update propagation:* Decide where to propogate which updates. Let all replicas receive all updates.
1
2
1
2
1
2
(c) *Scheduling:* Compute the ordering of updates.
1+2
1+2
1+2
(d) *Conflict resolution:* Transform conflicting updates to produce results intended by users.
(e) *Commitment:* Replicas agree on the final ordering. Their contents become consistent.

Figure 4: Replica uniformity by optimistic replication [Saito and Shapiro, 2002]

algorithms allow direct reads or writes of any replica most of the time, propagate updates in the background, and reconcile any update conflicts only after they occur (i.e., upon synchronization).

Optimistic replication couples the high availability of data with the ability to upgrade it while disconnected, to provide the workspace isolation required to host the cooperative copies necessary for concurrent and distributed software development. To be workable, data replication must be able to synchronize the updates made in isolation with the shared data under its charge. Hence, support for this kind of data alignment is often called synchronization technology [Pierce and Vouillon, 2004], and the resulting tools, synchronizers.

Under optimistic replication, a site, while being isolated from others, accumulates changes as they are *issued* (Figure 4*a*) until such time as the site detects it can (or should) communicate with another site. The eventual convergence, or uniformity, of content across all replicas is then established by a combination of the four mechanisms illustrated in Figure 4: update *propagation* (4*b*), update *scheduling* (4*c*), update *conflict detection and resolution* (4*d*), and update *commitment* (4*e*).

Reliable update propagation involves a site detecting when it can (or should) communicate with another, computing the set of changes to be transferred to make the two replicas consistent, and then transferring the changes quickly. In most optimistic replication systems, updates are eventually propagated to all sites. Since different sites may receive the changes of the same update in a different order, some sort of ordering policy is needed. Update scheduling establishes and enacts this policy. The scheduling algorithm must apply the updates quickly to reduce latency and increase concurrency, while trying to respect user intentions and avoid user confusion—goals that are somewhat contradictory.

Update conflict detection and resolution rely on merging. Many systems offer no conflict handling and will only propagate non-conflicting updates, and yet manage to deliver fair results. This can only work well if replica owners arbitrate their accesses to avoid conflicts in the first place—whether voluntarily or by policy. Fine-grained partitioning of data greatly aids in this regard [Magnusson et al., 1993]. Automatic conflict detection is desirable in that it frees users of having to mold their work patterns around the synchronizer or to partition designs unnaturally. Once a conflict is found, it must be resolved by reconciling the conflicting updates. Resolution strategies include the simple policy of "last write wins" and computing a semantic union of conflicting updates—not a trivial task.

The final stage of replication is update commitment. It is a mechanism for obtaining agreement of the sites on the scheduling and any conflict resolutions. Once obtained, committed updates can be applied to artifacts without fear of future rollback. Any logs and data structures associated with the update may be deleted at this time as well.

2.4.2 Replication Granularity

A replication system must work at the level of some minimal granule of update. File synchronizers typically synchronize directories composed of files, or files composed of records, while groupware systems regard a file as a document composed of elements such as sections, lines and characters. A system's conception of what constitutes its minimal granule is referred to as an *atom*.

Hence, exactly what constitutes an atom is relative to the particular kind of replication system. Furthermore, because systems can move between granule levels depending on the nature of their current task, what an atom is can vary over time. Some means of identifying an atom must also exist, e.g., path and file names for files, and line numbers for lines. Identification itself may be relative, which can add greatly to a system's complexity. This occurs when identity is context-dependent, as in a collaborative text editor where the location of a line might be numbered differently based on the context.

While tracking what is and is not an atom is expected of tools, users should not be burdened with this detail. People are much better at dealing with units of work or artifacts, these might be considered a person's minimal granule of shared data. Accordingly, the term *atom* is used when speaking of a minimal *granule of update* in some shared data, and *artifact* is used when considering a minimal *amount of shared data*. From the standpoint of a tool, atoms may be more important, but ultimately a tool must satisfy its users.

The relationship between atom, artifact, replica, and site is a hierarchical one. Namely, an atom is an updateable piece of an artifact; an artifact is stored at a site; a replica is a copy of an artifact that resides at some other site; and a site is some location in a repository, workspace, PC, PDA, etc. [Saito and Shapiro, 2002]. Since most optimistic replication algorithms manage each atom independently, focusing almost exclusively on them, the terms *replica* and *site* are sometimes treated with less care and used interchangeably.

2.5 Configuration Management and Merging

Configuration management (CM) is concerned with managing the evolution of large complex systems. The goal of CM is to "manage and control the numerous corrections, extensions, and adaptations that are applied to a system over its lifetime," and its objective is to "ensure a systematic and traceable software development process in which all changes are precisely managed" [Estublier et al., 2005].

When the systems are software, the concept is known as software configuration management (SCM), or in its earlier guise, *version control* [Grinter, 1995]. SCM is concerned with coordinating work and facilitating the interactions of multiple developers. What has set SCM apart from other applications of CM has been its emphasis on managing files and directories. Little has changed in this regard—the file system view is still heavily relied upon by most current software engineering tools in general, and it remains a core underpinning of SCM systems in particular [Estublier et al., 2005].

Concerned originally only with source code, SCM systems have grown to manage large data repositories that store all the kinds of artifacts that make up a software system. Over time, three key functionalities have come to be integrated into SCM systems: 1) the management of files involved in the creation of a software product, 2) the tracking of changes to these files as versions of the originals, and 3) support for the building of an executable system out of the managed files.

The tracking of historic versions of the entities under management, the so-called *configuration items*, gives SCM a strong dependency on merging. A merge is invoked, thus creating a new version, wherever corresponding replicas have undergone changes that result in their having conflicts.

2.5.1 Configuration Management Support

The support SCM provides to cooperative software projects may be divided into three major areas: product, tool, and process [Estublier et al., 2005]. This is a useful partitioning for looking at the history of SCM, and for understanding its successes and challenges.

Product support has been, from the beginning, a core function of SCM. The managing of the many files that comprise a software system, and the accumulation of changed versions breaks down along two technical dimensions:

- *Versioning.* The fundamental building block for all of SCM has substantial ramifications for merging and will be explored in the next subsection. A version is considered to be a set of *changes*, and each change in turn is considered to be a sequence of *change operations.* As system artifacts undergo modification, versioning maintains a historical archive of uniquely identified versions.

- *Configurations.* SCM systems support the aggregating of versioned items into higher-order configurations (which themselves may be versioned). Configurations keep item relations and properties clear, enforcing project consistency. Exactly which items of what versions are to make up a particular configuration are under user control.

Tool support facilities make configuration items available to other software engineering tools for interaction and manipulation. With this support, the SCM system becomes an abstraction layer on which other tools can operate and integrate their results. Two lines of major impact have been:

- *Workspace control.* As seen earlier, workspaces are a requirement for concurrent software development. They are also the primary mechanism by which access to configuration items is gained. SCM systems implement workspaces as isolation zones where developers can perform their day-to-day activities free from the interference of work being carried out at the same time in other workspaces, which may be distributed, or even disconnected. How the disparate workspace items are eventually integrated back into the SCM repository is an important consideration and almost always involves merging.

- *Building.* The build tools used to construct an executable program from a set of source files were initially developed independently of SCM systems. Today's SCM systems however, integrally support the building of the final software product. Obtaining the correct configuration items for a build requires precise control over the source items, which relies on the availability of accurate change information.

Process support has begun to make its way into SCM. Its responsibilities have evolved from managing only project files to managing of teams of collaborators engaged in software development and maintenance. Two areas of focus are:

- *Change control.* Restriction of developer access to configuration items (both read and update) is the root of all change control in SCM systems. Initial strategies were pessimistic and locking-based, but quickly gave way to optimistic and merging-based approaches. At some point, the capture of change rationale became a standard SCM feature.

- *Process control.* Originally, process control did not figure into what constituted an SCM system. The process was predefined by virtue of the way configuration items could be manipulated within the system, which was usually fixed. The push for process support has resulted in high-end SCM systems that allow organizations to design and enforce their own general development processes.

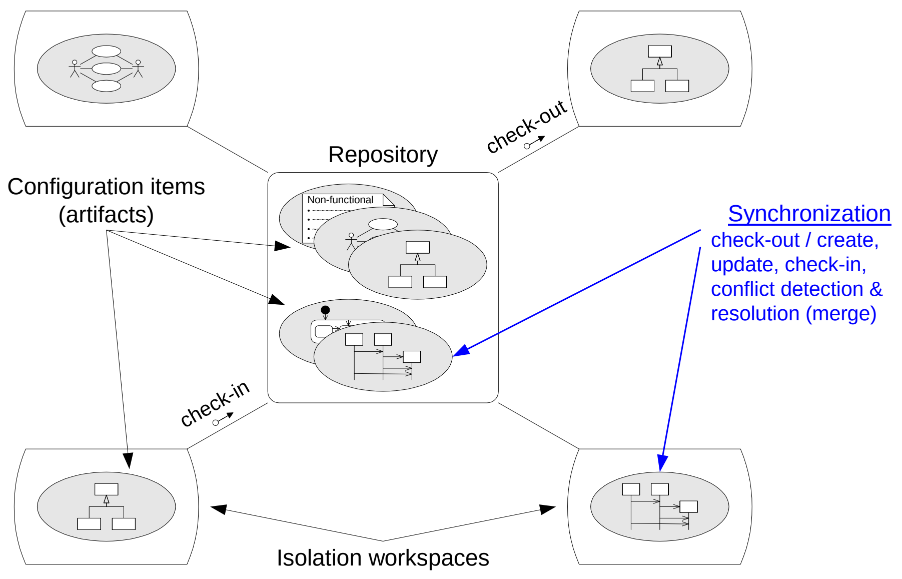

Repository
check-out
Configuration items (artifacts)
Non-functional
Synchronization
check-out / create, update, check-in, conflict detection & resolution (merge)
check-in
Isolation workspaces

Figure 5: Schematic of version control with synchronization and merge capability

2.5.2 Software Versioning

Software configuration management's answer to managing system lifecycle artifacts was to capture them as related configuration items, with access controlled through the classic revision/variant/merge version model [Estublier et al., 2005]. The first SCM tools supported software development with only source code versioning and a *time line* that tracked the evolution of a module through the code versions.

Configuration management procedures focused on controlling developers' abilities to alter code—the familiar *check-out* and *check-in* library metaphor [Grinter, 1995]: a new version of a module would be created whenever it was checked-out; the original would stay in the protected repository while changes were made to the copy; and finally, upon check-in, the original would recede down the time line while the altered copy became the new most current version of the module (Figure 5). This worked well and is still the basis of modern configuration management systems, but it did not take long for the limitations of the checked-out state to become apparent; the control was too rigid in that it prevented others from changing the same module at the same time, thereby slowing down the overall progress of work.

Later systems allowed for parallel development by not locking a developer out of a checked-out module, but instead by creating a new version for them from the last checked-in version. Whenever the inevitable collision of differing versions being returned to the repository occurred, a merge session was initiated. Performing the merge was usually a manual, and often painful, affair. Declining to do the merge would create a fork in the time line at that point. In theory, a fork could always be folded back into the main line by merging with the latest version of the module.

With accumulated practical experience and the advent of better visualization tools, the merging of simple textual artifacts has become more automated and less arduous. Still, the complexity of performing a merge rises significantly when the same lines of code or the same algorithm have been modified. Then, arriving at a resolution depends on an understanding of how the module works, and the merge session inevitably degenerates into a manually intensive process.

A form of versioning that has found only limited acceptance is called *change-set versioning*. In classic versioning, configuration items are first-class objects that are manipulated directly; the changes made to an item must be derived indirectly (e.g., by issuing a "diff"). Change-set versioning inverts this relationship, making changes the first-class objects and configuration items the indirectly derived ones. The approach stores each change as a delta, independent of all other changes. A new version of a configuration item is then constructed by applying a specified change-set or group of change-sets to the baseline version of the item.

The idea behind change-set SCM systems is the same as behind operation-based merging (Section 2.7)—the combining of change operations into an *operational trace*, which is then run against a baseline item to create a new (merged) version. The lack of interest in these systems is primarily due to the overabundance of choices they present: organizations find the systems overly complicated and too flexible for their needs [Estublier et al., 2005]. For a project with thousands of artifacts, the full flexibility of change-sets becomes unwieldy. Only a fraction of change-set combinations are actually useful—some combinations may not even parse or compile—and for certain item classes (e.g., binaries), there is no sensible way to combine deltas.

It was eventually understood that change-set functionality can be approximated by leveraging the difference and merge capabilities of classic SCM systems. Based on this insight, a hybrid approach proposed by Weber [1997] called *change-packages* offers a more practical alternative to change-sets. With change-packages, standard versioning technology, which is mature and efficient, powers the change management engine, while the flexibility of the change-set approach is held in reserve for those few cases where it is actually needed.

2.6 Merging Defined

The historic emphasis of SCM systems on files and directories sets a precedence that allows us to dispense with some generality. We assume, from this point forward, that merging will involve only files—in our case, models *or* change logs—and further, as commented on earlier, that only two files will be merged at a time.

To perform a merge is to somehow combine two replica-versions into a third version that takes into account the changes made to each. The third is created from contributions drawn potentially from both sides, i.e., the "left" and "right" inputs to the merge. Merging attempts to decide what to keep, what to leave behind, and what to replace, based on insights gathered from its inputs. Failing that, it must be able to garner and present enough information to elicit meaningful responses from a person asked to make the decision.

Two types of errors can result when performing a merge: inconsistencies and conflicts. An *inconsistency* occurs if an applied change fails because of an unmet precondition (e.g., changing a nonexistent element). A *conflict* is a contradictory pair of change operations, i.e., the two operations do not commute (e.g., giving different names to the same element). The *detection* and the *resolution* of conflicts go to the heart of the merge problem and are examined in more detail in Section 2.8.

In general, the algorithms used for *text merging* are simple heuristics that make no guarantees about correctness. Nevertheless, they have been found to usually produce reasonable results and to be very useful in practice [Conradi and Westfechtel, 1998]. Since the emergence of traditional line comparison techniques for the differencing and merging of ASCII text, research has sought to supply better functionality by rooting techniques in strong theoretical foundations. Research has followed two tracks: incorporating semantics for more accurate and wide-ranging differencing and merging, and extending differencing and merging to handle binary artifacts.

Despite many desirable characteristics, semantics-based merging has had almost no impact on SCM systems [Estublier et al., 2005]. Virtually all leading SCM systems, however, can detect block moves and compress any kind of file, giving them the ability to difference and merge binary artifacts. It remains to be seen whether interest in

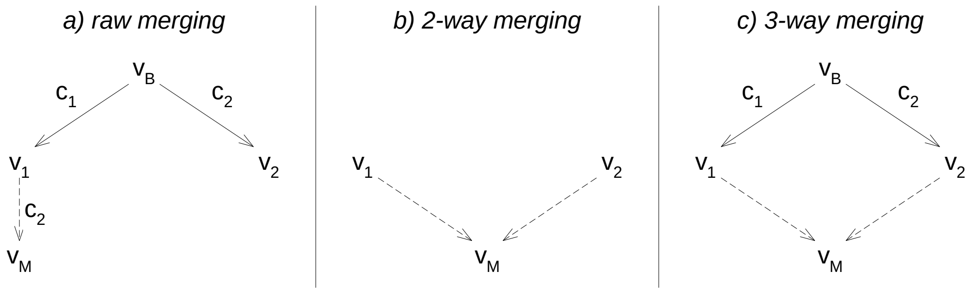


Figure 6: Types of merging [Conradi and Westfechtel, 1998]

model-driven engineering will precipitate investigations along the other track.

2.6.1 Merge Types

All merges can be categorized as one of three fundamental types: raw, two-way, or three-way merges [Conradi and Westfechtel, 1998]. Figure 1 provides a schematic representation of each.

Raw merging applies a change to a context different from the one in which it was made. In Figure 1a, change c_1, and at a later time, change c_2 are executed independently of each other on replicas of base version v_B to produce, respectively, versions v_1 and v_2. To effect a raw merge, changes are applied to an evolving merged version in the order of their original executions. Since the first change was c_1, its replica-version v_1 becomes the base for future changes. The second change c_2, is then applied to this new base to produce the final merged version v_M.

If c_1 alters something that c_2 depends on, the merge may fail with an inconsistency. In raw merging, no thought is given to reordering changes to avoid such eventualities, so failures are common. Conflicts, however, do not arise. Where there are any contradictions, the last applied change of a conflicting pair simply overwrites what has gone before it.

A *two-way merge* as shown in Figure 1b compares the state of two, perhaps unrelated, versions v_1 and v_2, and combines them in some fashion into a single version v_M. While a two-way merge can automatically avoid inconsistencies by rearranging change operations, it can detect, but not resolve, except arbitrarily, a whole class of *add-delete* conflicts. The problem occurs when an element present in one version is not found in the other. Without the originating source it is impossible to tell if the element was added to one version, or deleted from the other. This is the fundamental difficulty with two-way merges: lacking baseline data or a historical trace of changes, there is insufficient information for deciding how to resolve a conflict [Mens, 2002].

The fact that *three-way merging* can resolve the add-delete problem is why almost all present-day merge tools use it. The only difference from two-way merging is access to a baseline version that is the common ancestor of the versions to be merged. Now when there is an added or missing element, the base v_B in Figure 1c may be consulted to determine if the causal change was made in v_1 or v_2. Conflicts caused by changes made to both replicas, however, can still arise and are resolved just as in a two-way merge: heuristically, arbitrarily, or by seeking user input.

2.6.2 Merge Level

Besides type, merges may be further characterized by level. Level gives an indication of what knowledge of the product space a merge must possess in order to carry out the operation successfully [Conradi and Westfechtel, 1998]. In other words, it declares at what semantic level the merging is to be performed. The levels, in order of increasing difficulty of implementation are: textual, syntactic, and semantic.

Textual merging is used for merging character-based files. The algorithm's working element is usually a file line, or record. This limited view means that file comparison works on a line-by-line basis when detecting conflicting characters or strings. As there is little basis for deciding the resolution of a conflict, the results of the merge can appear somewhat arbitrary, but in practice, textual merging works quite well. This is not because of any algorithm, but mainly due to policies that reduce the possibility of conflicts by enforcing a fine granularity of work items, and the coordination of changes. There are, however, well-known limitations to textual merging [Mens, 2002].

Syntactic merging exploits knowledge of the syntax of the language that the files' content is expressed in to make more intelligent choices when resolving conflicts between the files being merged. Thus, the merge algorithm is required to know the grammar of the artifacts to be merged in detail. For this reason, it is usually restricted to working with one language or family of related languages. Additionally, the results can be guaranteed to be syntactically correct only if the grammar is context-free.

In addition to syntax, *semantic merging* takes the semantics of the items to be merged into account, which involves dealing with language-level conflicts. Not only would this be quite helpful for text-only merges, it would prove beneficial for highly structured data, like models, too. A semantic merge tool must perform a sophisticated analysis in order to detect and decide how to resolve conflicts. Unfortunately, for most domains, coming up with a definition of semantic conflict that is neither too strong nor too weak, yet decidable, is a hard problem [Conradi and Westfechtel, 1998].

2.7 Merging Approaches

Approaches taken to merging fall into two broad categories: those that are state-based and rely on a static view of the replicas being merged; and those that are operation-based and analyze a dynamic trace of user actions, or change operations, gathered during replica modification. Both categories have been touched upon: two-way and three-way merging were explained in terms of state; and consistency criteria, change set SCM systems, and raw merging in terms of operations.

2.7.1 State-based Merging

If it relies solely on a comparison of the final states of replica-versions, a merge tool is said to be *state-based*. It makes no use of information concerning the actual evolution of an artifact, but relies only on the difference in state between representations of that artifact. The schematic of Figure 7*a* depicts the merging of *left* and *right* versions of a common source after extraction of their differences.

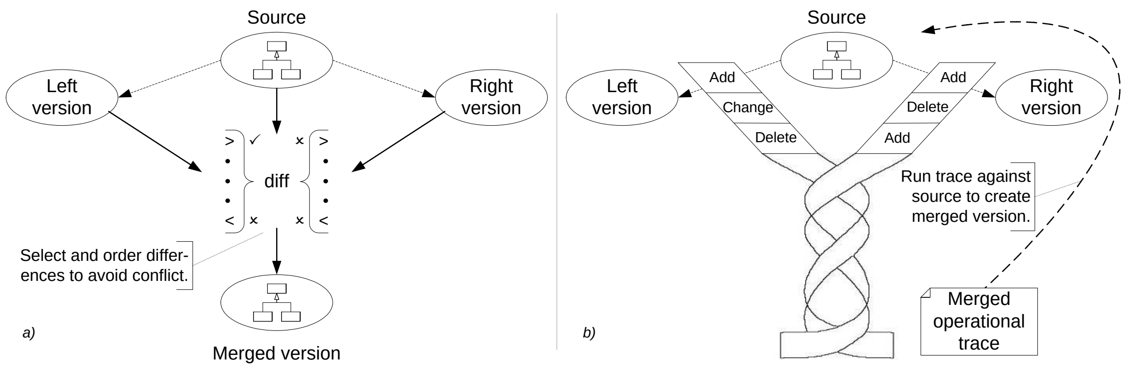


Figure 7: State-based and operation-based three-way merges

These differences comprise the changes made to the replicas. They are selected for their contributions to a more complete merge, forming an update set that is eventually applied to the source version. As many of the changes as possible are assimilated by the set, with those that induce inconsistencies or conflicts being omitted. For these, special algorithms are invoked to remove the merge errors, or, as a last resort, to ask the user to do so. State-based merging:

- Uses simpler algorithms, making tools easier to implement.
- Has no need for a change recording, or logging, unit.
- Requires no knowledge about how a version may have reached its final state.
- Works at high levels of change granularity.
- Provides fewer options for resolving merge errors.
- Does not need to handle redundant changes.
- Has complete state information.
- Has good performance.

Due mainly to easier implementation, tools for merging have been predominantly state-based, though interest in operation-based merging is growing [Bartelt, 2008], [Koegel et al., 2009].

2.7.2 Operation-based Merging

Rather than having to reconstruct its updates from artifact difference as a state-based tool must do, a history, or *operation-based*, tool has access to the historical traces of change operations executed in modifying the replica-versions to be merged. The operations are of a finer grain than those produced by a straightforward "diff" of artifact states.

The operations from two sequences of model changes are being intertwined in the schematic of Figure 7*b* to create a merged operational trace. To avoid or eliminate merge errors, operations may be included or omitted from the final trace, kept in order, reordered, or interleaved with changes from the other side. The resulting merged trace is then run against the common source to generate the merged version of the inputs. Operation-based merging:

- Uses more complicated algorithms, making tools harder to implement.
- Requires tight coupling with a change recording unit (which may not exist).
- Can use knowledge of operation execution to reason about changes.
- Works at low levels of change granularity.
- Provides more options for resolving merge errors.
- Results in more intuitive merge session interaction.
- Must deal with redundant changes.
- Has incomplete state information.
- May suffer from performance issues.

The proposed distributed versioning model for the OMG's metaobject facility [Hnětynka and Plášil, 2004] holds promise for making change history more readily available to designers of operation-based merge tools.

2.8 Merging Errors

Merging works by applying a series of updates in the form of differences (for state-based) or change operations (for operation-based) to the version that is to be gradually transformed into the merged outcome. As already alluded to, an applied update can fail, either because one of its preconditions is not met (an inconsistency), or because it contradicts a previous applied update (a conflict).

Inconsistencies are detected either beforehand by a dependency analysis of the pending updates or after the fact by an actual failure. A merge algorithm that experiences a failure after application of an update can backtrack to a point before the failure in order to make a repair. Repairs consist of reordering application of the updates so as to satisfy the failing update's preconditions. Unless there is a cycle in the dependency graph, an inconsistency can always be repaired in this fashion.

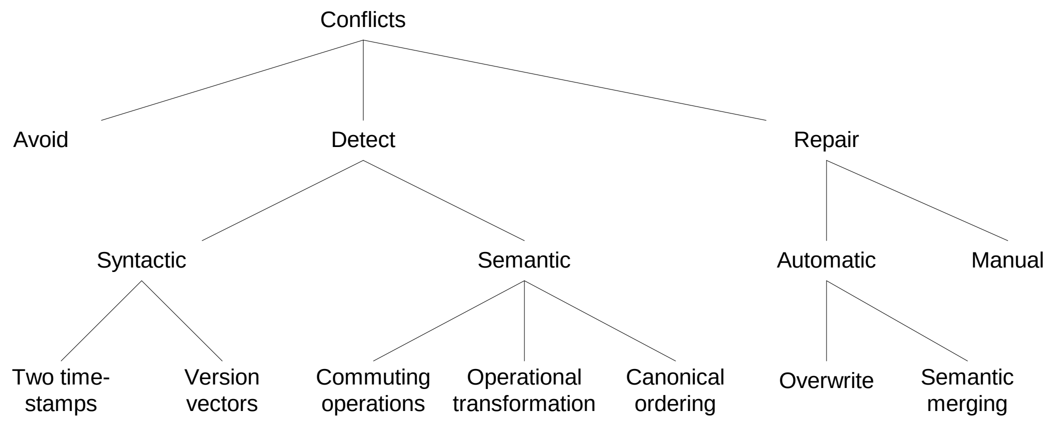


Figure 8: Taxonomy of conflict handling techniques [Saito and Shapiro, 2002]

No such simple solution exists for conflicts, however. The taxonomy of Figure 8 presents the known techniques for handling conflicts, grouped by overall strategy: *avoid*, *detect*, or *repair*. Nothing is found under the avoid strategy, because avoidance is a question of process, not technique. The techniques under the other two strategies are further broken down into resolution types.

Syntactic conflict detection is not particularly suited to the highly structured data of models. The techniques of *two timestamps* and *version vectors* both use the notion of a *happened-before* relation, a means of ordering in concurrent systems based on the causal relationship of event pairs. A conflict is flagged if an update violates the happened-before relation it has with *a prior* update (i.e., it is applied out of order).

Semantic conflict detection, by considering the meaning of an update, filters out those syntactic conflicts that, according to the semantics of the item being updated, are not actually in conflict. Being difficult to implement, synchronizers and mergers that offer semantic conflict detection are rare. The detection techniques pictured under the semantic category are concerned with the scheduling of update operations; that is, given a log or trace of updates to be performed, they try to recognize problems in the update ordering.

Canonical ordering presumes a standard order for the kinds of updates being applied (e.g., lexicographic order for strings) or the policies under which they are being applied. This is of limited use in collaborative settings, because most processes cannot define such an ordering for the changes that modify their work items.

Operational transformation (OT) technology supports a range of functionalities for consistency maintenance and concurrency control in the collaborative editing of, primarily, documents. The basic idea is to transform (or adjust) the parameters of a later operation, based on the effects of previously executed concurrent operations, such that the transformed operation can achieve the correct effect and maintain document consistency. Detection involves knowing what operations can occur and how they interact, so effects of earlier operations can be countered by transforming the current operation. OT works well in groupware applications where updates are restricted to text-based edits of documents. Again, highly structured data are problematic.

The last technique, *commuting operations*, is a viable way of detecting contradictory pairs of updates. If the outcome of applying two updates to an item is dependent on their order, then the operations are not commutative and, hence, conflict. In addition to its intuitive appeal, the technique is fairly straightforward to implement. It is taken up again in the next section in the context of model transformations.

Any type of conflict detection is predicated on correctly matching up, or mapping, elements from one artifact to another. If a tool fails to realize that a pair of elements from corresponding artifacts represent the same thing, then it will not be able to detect when updates made to both are contradictory. This problem is most notorious in tools that rely solely, as many do, on unique identifiers for matching, and is considered in detail in Section 5.2.2.

Once detected, a conflict can be ignored or repaired. Tools that ignore conflicts may simply halt with an error or produce inconsistent outcomes, which must then be fixed with some sort of post-processing. Though not entirely desirable, this may be workable for some domains. Normally, however, a repair will be attempted, more often than not, by eliciting the manual intervention of the user.

An automatic *overwrite* applies one change of a contradictory pair, chosen either arbitrarily—usually the last made—or from a predefined default, or "master," side of the merge, over the top of the other. Semantic merging, of course, tries to bring meaning into the repair process, but if semantic detection is rare, semantic merging is even more so. Enabling semantic detection and repair with decision tables has been a major focus of this research and is the topic of Subsection 7.4.3.

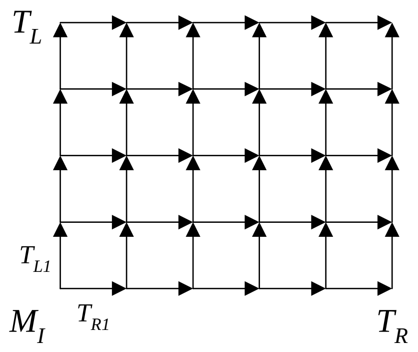


Figure 9: Grid of model subtransformations

2.9 The Merge as a Model Transformation

If we now consider only models, and the changes that may be applied to them: each change applied to a given model can be thought of as a function, or an operation, on that model that yields a new model. Or, alternately, as a *transformation* that takes the model and transforms it into another version of itself. A sequence of such operations then, may be functionally composed into a single transformation T. Applying T to some initial model M_I conforming to some metamodel MM_I will then produce a final model M_F conforming to some metamodel MM_F:

$TM_I = (T_1 \circ \ldots \circ T_{n-1} \circ T_n)M_I = M_F$,

where T_i is one in a sequence of *subtransformations* of T. Individual pairs of subtransformations may or may not commute. If $MM_I = MM_F$, then the transformation is *endogenous*, otherwise it is an *exogenous* transformation.

A three-way merge involves two transformations: one that it is convenient to think of as being on the left T_L, that transforms a replica of the initial base model M_I into model M_L; and one on the right T_R, that transforms another replica of the base into model M_R. An operation-based merge of models then, can be considered as a particular path traced out on a grid of subtransformations (Figure 9) of the left and right models [Lippe and van Oosterom, 1992].

The paths of this *transformation grid* that respect the original ordering of the involved subtransformations (e.g., T_{L_i} comes before $T_{L_{i+1}}$, even if interleaved by T_{R_j}) are known as *weaves*. Those paths which are a permutation of the union of subtransformations are *full* weaves, while those composed of fewer subtransformations are *partial.* To trace a path from the origin to another node is to execute the sequence of elementary operations that lie on the path, and thus transform M_I into a new model.

The discussion on commuting operations in the last section lets us conclude that if two subtransformations do not commute, they are in conflict. All weaves that lead to a particular node contain the same subtransformations and differ only in their interleavings. Any non-commuting pairs in those weaves will thus cause the models produced to be different. More generally, if the subtransformations T_1 and T_2 commute globally, i.e., $T_1 \circ T_2 = T_2 \circ T_1$, then they do not conflict. However, transformations that do not commute globally *may* commute locally when operating on disjoint sections of a model, that is, $(T_1 \circ T_2)M = (T_2 \circ T_1)M$, and consequently will not conflict when transforming M. To test for commutativity, Lippe and van Oosterom [1992] define a *before* predicate for precedence. Briefly, if operation a must come before operation b, then $\mathit{before}(a, b)$ will evaluate to *true*. If $\mathit{before}(b, a)$ also evaluates to *true*, then a conflict exists between the pair of operations.

In the context of transformations, then, the essence of merging becomes to blend T_L and T_R into a single transformation that will transform initial model M_I into a final merged model M_F. In the absence of conflicts, the transformations may be combined in whole tip-to-tail fashion—as is done with vectors—otherwise they must be cobbled together piecemeal, subtransformation by subtransformation, which is precisely what operation-based merging attempts to do.

The topic of "the merge as a model transformation" is taken up again in Chapter 4, where the notion is repeatedly put to work on the surface of a plane in order to build up a merge space. But first, as a means of gaining an appreciation for the difficulties and problems with model merging, an examination of its operation in a selection of current MDE tools is made.

Chapter 3

Model Merging and State-of-the-Art Tools

Committing programmer source code changes to a common repository is such an ingrained, day-to-day part of software development that it rarely elicits comment. Even when updates clash and conflicts must be resolved there is little complaint. This is enhanced by workplace policies that avoid many conflicts in the first place. The clear mapping of code modules to files and our awareness of their text-based nature lead us to accept demands for human input, and less than perfect merges as being perfectly reasonable.

While our source code experiences may prepare us to some extent, there are surprises to be had in the merging of models. This section affords a look at merging through the eyes, so to speak, of a small selection of current modeling tools. It demonstrates some of the issues raised in the Problem Statement (Section 1.1) and gives a better appreciation of the difficulties facing model merging.

3.1 Merging in Practice

A full MDE merge cycle can be reduced to three high-level steps: replicate, elaborate, and merge. Assuming that the merging will be three-way and, again, that we need only concern ourselves with two replicas, then a pair of developers will, in the roles of *Modelers*: 1) replicate a copy of a common base model for themselves, 2) elaborate their respective replicas in isolation creating new versions of the base model, and 3)

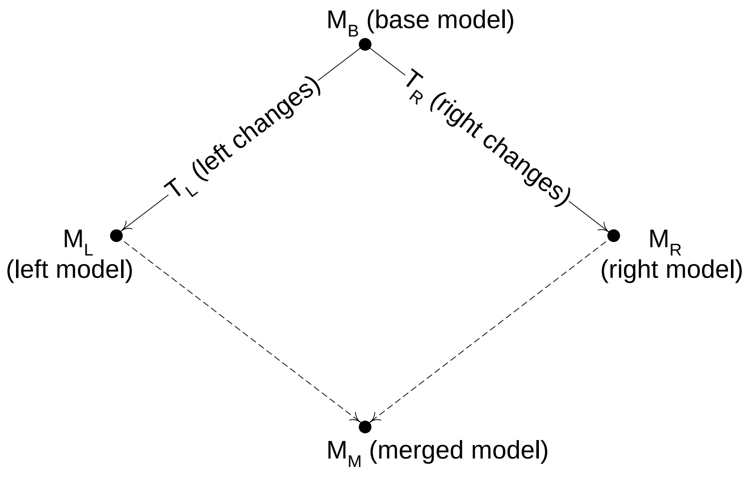


Figure 10: Merging two replicas derived from a common base model

then take on the role of *Merger*—most likely jointly—to bring the replica-versions into alignment, producing a merged version of the three models.

The merge diamond in Figure 10 is a schematic of the merge cycle, representing one *merge session* of potentially many. A portion of a chain of such sessions that emerge over the course of development is pictured in Figure 11a, where the merged output M_{M_n} of a replicate-elaborate-merge cycle becomes the base $M_{B_{n+1}}$ input of the cycle immediately following it. This is an idealized view, because it incorporates all of the changes of both Modelers. More realistically, due to conflicts, the Modelers will find themselves acting as the Merger, and having to spell out to the tool details of the models' structure and semantics of their changes.

The truncated cycles of Figure 11b reflect the partial nature of automatic merging in an environment conducive to conflicts. If M'_{L_n} and M'_{R_n} are at the limit of conflict-free merging in cycle n, then M'_{M_n} is as far as automatic merging can proceed with full confidence. From here, a human merger must conceptually spring from the partially merged model M'_{M_n} to assist the merge in bridging the gap to M_{M_n}, in order that a new cycle $n+1$ can be started off the newly formed base $M_{B_{n+1}}$. The more solid and complete the tool can make the partial model, the narrower the gap and the easier it will be to bridge.

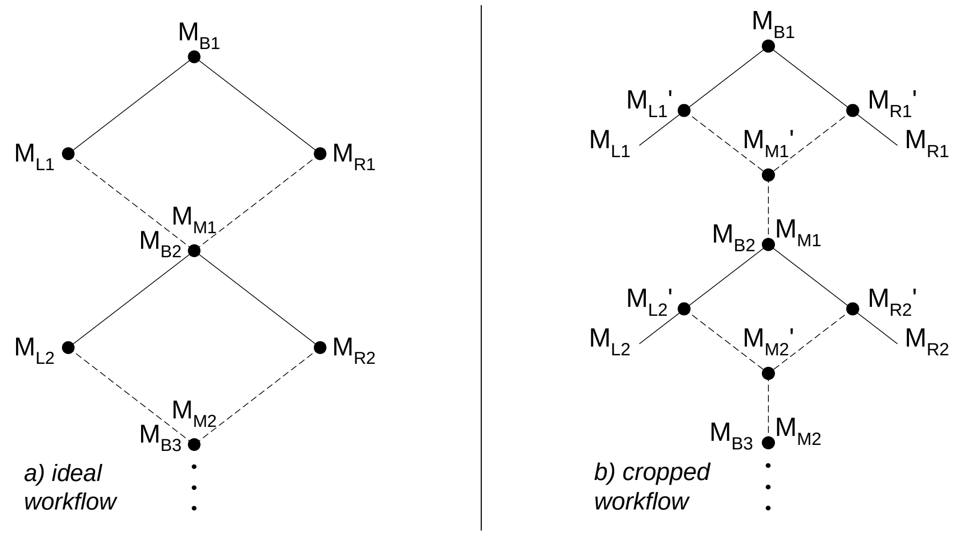


Figure 11: Merge workflow of repeating replicate-elaborate-merge cycles

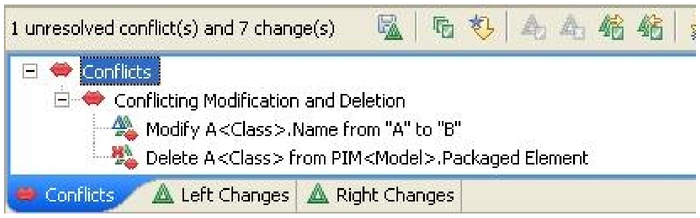
3.2 Merge Capabilities of Select Modeling Tools

To identify merge shortcomings, an evaluation of the capabilities of three state-of-the-art modeling tools was done. The tools were selected for differences in their merging approaches, as well as in their merge and conflict presentations. Two of the choices are Eclipse-based and partially rely on its Compare Facility [EMF Compare, 2010]. Not surprisingly, they have a similar look and feel, but their outcomes *do* differ. And though two are from the same vendor, they are quite different in operation and results.

The tools all support modeling with UML 1.4 or later, and were set up to run under Sun's Java Runtime Environment SE 5.0. What a tool displays and how it is displayed act to constrain what merge decisions can be made: The "granularity" of a tool's display determines exactly what a Merger has control over, while the layout and the amount of UI "noise" generated affect their comprehension of the merge. In what follows, we adopt the convention used by many tools: referring to one of the replica-versions input to the merge process as the *left* model, and the other as the *right* model; visually laying them out side by side, in the corresponding positions.

3.2.1 IBM Rational Software Architect

We investigated model merging with IBM Rational Software Architect 7.0.0.2 (RSA) running as a plug-in of Eclipse 3.1 on top of Java 5.0. RSA is capable of both two-


1 unresolved conflict(s) and 7 change(s)

```
Conflicts
    Conflicting Modification and Deletion
        Modify A<Class>.Name from "A" to "B"
        Delete A<Class> from PIM<Model>.Packaged Element
```

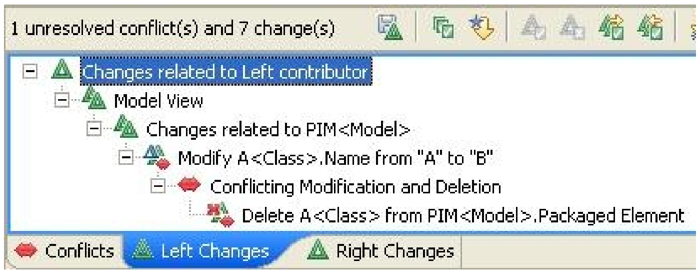
Figure 12: Model conflicts in Rational Software Architect

and three-way merges, but we only evaluated three-way merging.

Merging in RSA follows a compare, resolve, and (optionally) validate cycle. First, the three models are compared against each other. The tool then notes differences between the replica-versions. Lastly, any conflicts between the left and right models brought about by the changes are displayed in a *Conflict Pane*. The changes of a conflicting pair are displayed as leaves on their own branch in a tree of conflicts. In Figure 12 the right change is attempting to delete a class that the left change is trying to rename. To resolve a conflict, the Merger can accept or reject either side's change (implicitly choosing the opposite action for the other change), or ignore both (i.e., keep the base's original state).

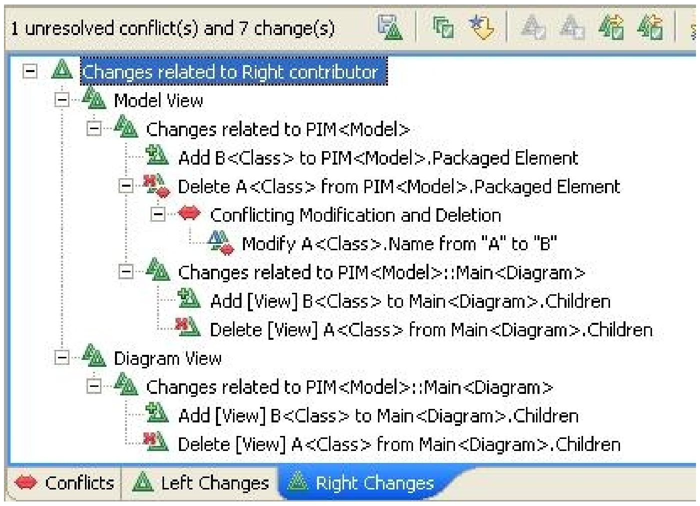
To better understand a conflict, it is often helpful to view the two sets of conflicting changes within the context in which they were made. Therefore, RSA shows the changes made to the models in a pair of *Change Panes*, one dedicated to either side (Figures 13 and 14). The information given in the conflict pane and in the two change panes overlaps, but fortunately they are linked, so that resolution actions taken in any one of the panes are reflected in the other two.

The *Right Change Pane* (Figure 14) illustrates the somewhat confusing distinction that RSA insists on enforcing between model and view (even as it refers to both as


1 unresolved conflict(s) and 7 change(s)

```
Changes related to Left contributor
    Model View
        Changes related to PIM<Model>
            Modify A<Class>.Name from "A" to "B"
                Conflicting Modification and Deletion
                    Delete A<Class> from PIM<Model>.Packaged Element
```

Figure 13: Left model changes in Rational Software Architect


1 unresolved conflict(s) and 7 change(s)

```
Changes related to Right contributor
  Model View
    Changes related to PIM<Model>
      Add B<Class> to PIM<Model>.Packaged Element
      Delete A<Class> from PIM<Model>.Packaged Element
        Conflicting Modification and Deletion
          Modify A<Class>.Name from "A" to "B"
      Changes related to PIM<Model>::Main<Diagram>
        Add [View] B<Class> to Main<Diagram>.Children
        Delete [View] A<Class> from Main<Diagram>.Children
  Diagram View
    Changes related to PIM<Model>::Main<Diagram>
      Add [View] B<Class> to Main<Diagram>.Children
      Delete [View] A<Class> from Main<Diagram>.Children
```

Figure 14: Right model changes in Rational Software Architect

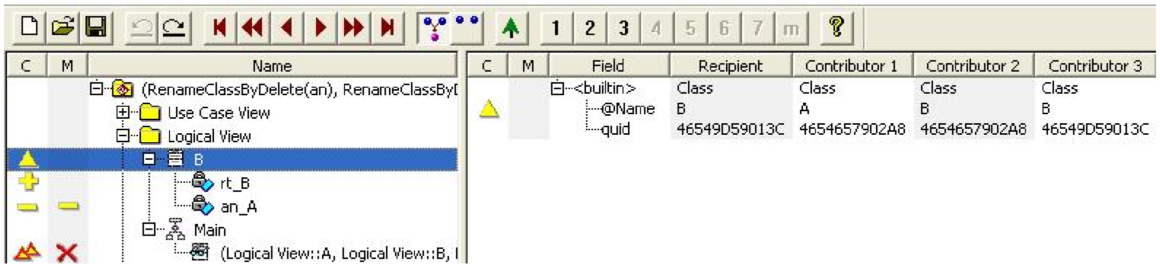
a "View"). Observe that the fact a Modeler has added a class **B** to the right model is noted in *three* separate places in the same pane, *two* of them having to do with the view ("Add [View] B"). If this change had also resulted in a conflict, the class addition would have been captured at least once more in each of the other two panes. This level of *cognitive noise* can, as we will see, lead to some undesirable results.

Once all conflicts have been resolved, the merged outcome can be saved and validated. Unfortunately, validation cannot be done before saving; overwriting one of the replica-versions in the process.

3.2.2 IBM Rational Rose

Though also offered by IBM, Rational Rose 7.0.0.0 (Rose) follows in the footsteps of Rational's classic line of modeling tools. The Rose Modeler and its Model Integrator are stand-alone tools running on Java 5.0 in our installation. The Integrator allows for specifying an n-way merge, but again, we only considered three-way merging.

The first model specified to the Integrator is assigned the role of *base* (i.e., the common ancestor) and given the name *Contributor 1*. The left and right models (when input in that order) become *Contributors 2* and *3*, respectively (right side of Figure 15). In Compare Mode, Rose only shows the differences between the participants, while in Merge Mode it produces a merged outcome named *Recipient*. The Merger is free to alter the choices made by the tool for Recipient by overriding them



C	M	Name
		(RenameClassByDelete(an), RenameClassBy[
		Use Case View
		Logical View
⚠		B
✚		rt_B
▬	▬	an_A
		Main
⚠	✖	(Logical View::A, Logical View::B, l

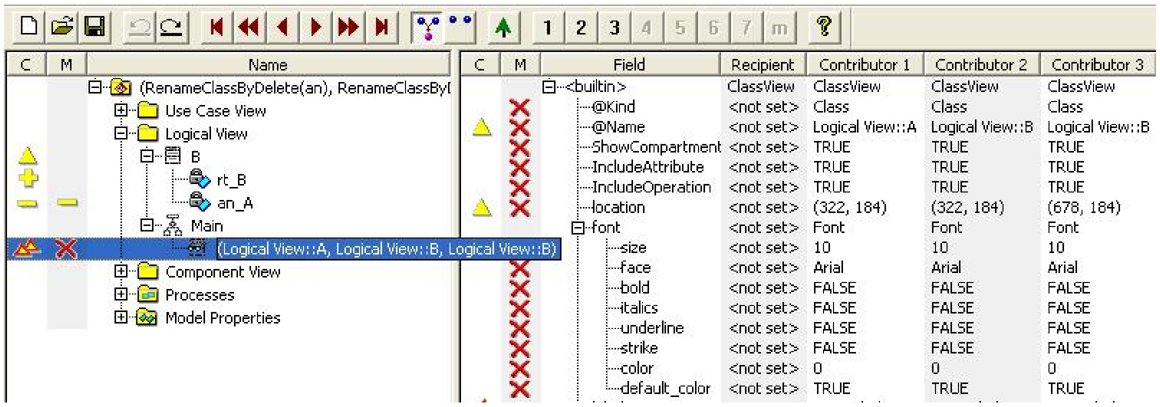
C	M	Field	Recipient	Contributor 1	Contributor 2	Contributor 3
		<builtin>	Class	Class	Class	Class
⚠		@Name	B	A	B	B
		quid	46549D59013C	4654657902A8	4654657902A8	46549D59013C

Figure 15: Model changes and conflicts in Rational Rose

with individual contributions from among the input models.

Like RSA, Rose also maintains a distinction between model and view. The highlighted item in Figure 15 shows *model* attributes for class **B**; selecting items from under the "Main" branch in the tree will reveal class **B**'s associated *view* layout details (Figure 16). The icons in the left margins indicate model changes (column 'C') and merge conflicts (column 'M'). The display is less intuitive than RSA's. For example, some investigation is required to uncover that the flagged conflict is caused by the left and right both having added a class named **B**. That being said, RSA misses this potential conflict altogether.

Rose follows the same compare models, resolve conflicts, and (optionally) validate cycle that RSA does. Everything is presented in the two panes of the main window. Item changes, additions, deletions and conflicts are presented hierarchically in the left pane, while contributor details about the selected item are presented in the right. Making no decisions with regard to an item, implicitly accepts Rose's merge choices for that item. Conflicts, however, *must* be manually resolved by the Merger. This



C	M	Name
		(RenameClassByDelete(an), RenameClassBy[
		Use Case View
		Logical View
⚠		B
✚		rt_B
▬	▬	an_A
		Main
⚠	✖	(Logical View::A, Logical View::B, Logical View::B)
		Component View
		Processes
		Model Properties

C	M	Field	Recipient	Contributor 1	Contributor 2	Contributor 3
		<builtin>	ClassView	ClassView	ClassView	ClassView
	✖	@Kind	<not set>	Class	Class	Class
⚠	✖	@Name	<not set>	Logical View::A	Logical View::B	Logical View::B
	✖	ShowCompartment	<not set>	TRUE	TRUE	TRUE
	✖	IncludeAttribute	<not set>	TRUE	TRUE	TRUE
	✖	IncludeOperation	<not set>	TRUE	TRUE	TRUE
⚠	✖	location	<not set>	(322, 184)	(322, 184)	(678, 184)
		font	<not set>	Font	Font	Font
	✖	size	<not set>	10	10	10
	✖	face	<not set>	Arial	Arial	Arial
	✖	bold	<not set>	FALSE	FALSE	FALSE
	✖	italics	<not set>	FALSE	FALSE	FALSE
	✖	underline	<not set>	FALSE	FALSE	FALSE
	✖	strike	<not set>	FALSE	FALSE	FALSE
	✖	color	<not set>	0	0	0
	✖	default_color	<not set>	TRUE	TRUE	TRUE

Figure 16: View changes and conflicts in Rational Rose

involves choosing one of the contributions: selecting Contributor 1 will restore the element to the base model's original state, while selecting Contributor 2 or 3 will take the contribution from (in our setup) the left or right model, respectively.

3.2.3 Sparx Enterprise Architect

Sparx System's Enterprise Architect 6.5 (EA) is also a stand-alone tool that requires Java 5.0 to run. Unlike the other tools, EA is only capable of two-way consolidation. Models may only be synchronized in EA. Once started, synchronization proceeds automatically, resolving conflicts as needed without input from the Merger. As expected with synchronization, after the operation, the participating models will be identical. The synchronization follows two simple rules: additions are cumulative—anything added to one model will end up in both; and deletions prevail over modifications—common elements deleted from one model are removed from both, *even if modified on the other side*. The tool appears to make no distinction between model and view.

Because it stores discarded changes, EA has the potential to let the Merger resolve conflicts differently than the synchronizer might. However, it is up to the Merger to inspect the *Resolve Conflict Dialog* after synchronization to see how any conflicting situations have been acted upon. A conflict in EA can only arise in the context of the second synchronization rule—when an element has been deleted. This means that adding the same element on both sides will produce duplicate elements in the outcome.

3.3 Simple Merges

The tools overviewed in the last section are evaluated here with scripts of hypothetical merge sessions that typify situations that can easily arise in any real-world, collaborative modeling effort. Using extremely simple models, the *primitive evaluation cases* uncover interesting and, in some cases, questionable behavior on the part of the tools. The cases are primitive in the sense that each one packages a single concept into a unit that aims to minimize distractions and side effects. Each merge evaluation case strives to isolate some aspect of real-world models that, upon merging, can lead to surprising and even counterintuitive outcomes.

An evaluation case describes a collection of possible paths through a merge session. Each path—referred to as a *merge script*—details the changes to apply to the two replicas before they are to be merged, and then the merge actions to take with respect to each change within the tool doing the merge. It is the primitive scripts that do not deliver the expected outcome (i.e., those that exhibit counterintuitive behavior) that are described here.

Only three-way merges are investigated. For Enterprise Architect, a three-way merge is “simulated” via two, two-way merges—base with left, and merged base with right. Although this work-around does not solve the add-delete problem, it *does* involve all three models in the merge.

3.3.1 Vendor Tools

The tools investigated make different assumptions and operate from different frames of reference. If execution of a merge script is to be comparable across all the tools, a common specification must be adopted.

An evaluation case is documented using an initiating figure and a family of execution figures—one for each tool tested. The initiation part (e.g., Figure 17) shows the three input models with which to set up the merge session. The base model (Contributor 1 in Rose parlance) is placed at the top of the figure, and the left and right models (Contributors 2 and 3) at its bottom left and right, respectively. This pyramid arrangement reflects the relationships between the models.

The execution part (e.g., Figure 18) details how the hypothetical merge session is to proceed. It lists the left and right replica changes on their respective sides, along with the merge actions to be taken with regard to each change. A list of such *change-action pairs* forms a merge script. Beneath the script is the merge outcome obtained from the tool in question, as rendered by that tool. Again, the layout visually echoes the relationship between the left and right models.

Merge actions are expressed using the operations and arguments of Table 1. In RSA, `reject` is directly available, but must be accomplished in Rose by explicitly accepting the pre-change state of the base model. The `sync` operation applies only to Enterprise Architect—the only operation the tool is capable of. Recall that synchronization results in the participating models becoming identical. For simplicity's

Table 1: Tool merge actions

Operation	**Arguments**	**Description**	**Notes**
`accept`	**m**odel, **v**iew	Accept specified component	Not EA
`reject`	**m**odel, **v**iew	Reject specified component	Not EA, Rose #1
`ignore`		Do nothing about change	Not EA
`sync`	**B**ase, **L**eft, **R**ight	Synchronize specified models	EA only

sake, all of the merge actions appearing on the left are performed before any of the actions on the right.

Most modeling tools are concerned with the model *and* its views. When this crosses over into merging, it complicates the operation. If we assume that there is one model component and one view component, then a merge action can be applied to none, either, or both. The arguments m for model and v for view indicate which component(s) an operation is to act upon. Similarly, an argument to the `sync` operation specifies which model is to be synchronized with the model that invokes the operation: B, L and R being the base, left, and right model, respectively.

Ultimately, the choices presented, the conflicts identified, and the corresponding actions that are allowed to the Merger are determined by the tool's capabilities and its merge philosophy. Consequently, in order to satisfy the *intent* of a particular merge script, the script's realization will necessarily vary from tool to tool.

Add Same Class

In the first primitive, as seen in Figure 17, both Modelers add an empty class **B** to their replicas of the ancestor, which consists of a single empty class **A**. The RSA script of Figure 18 instructs the Merger to accept the added class from each side. Since RSA insists on carrying its notion of model and view from its design environment into the

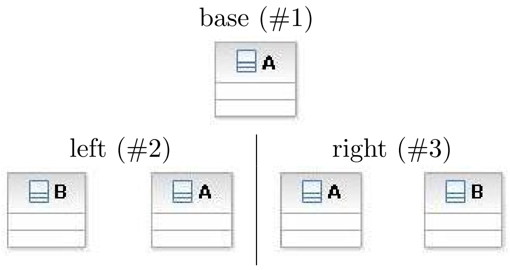


Figure 17: "Add same class" common base and evolved replica-versions

add **B**: `accept(m,v)` | add **B**: `accept(m,v)`

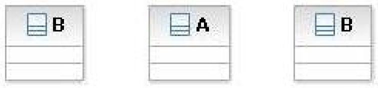


Figure 18: "Add same class" script and outcome in Rational Software Architect

merge process, the script makes sure to include the effects of the change as manifested in *both* components by passing the m and v arguments to the `accept` operations.

The resultant model is rather counterintuitive. It is unlikely that, when accepting both additions, the Merger was envisioning a model containing *two* classes with the same name. Not only did the tool fail to raise any red flags, a post-merge model validation by the tool found no problems with this state of affairs. This so-called *duplicate element problem* is a common one with merge tools that rely exclusively on ID for element matching.

Enterprise Architect suffers from the same deficiency, as revealed by Figure 19. Here, two different scripts yield the same outcome. Both scripts attempt to mimic a three-way merge: the first synchronizes the left model with the base (making them identical), then that result with the right model. The second script shortcuts this double `sync` by directly synchronizing the left model with the right. In either case, the tool detects no conflicts.

Rose takes effort to preclude having two classes with the same name appear in the merged model. Instead the classes are considered to be the same model element, even though they have different IDs. The tool resolves the logical class duplication automatically by defaulting to the right's version (this can be overridden), but it finds a conflict in the view of the merged model: where should the surviving class **B** be placed? Where left's class was, or where right's class was? Though trivial from the standpoint of a complete and consistent model, this predilection for burdening model merging with view details is a frequent source of confusion. To keep things

add **B**: `sync(B)` | add **B**: `sync(B)`

or

add **B**: `sync(R)` | add **B**: `ignore`

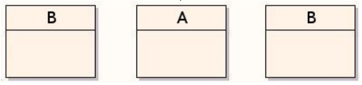


Figure 19: "Add same class" scripts and outcome in Enterprise Architect

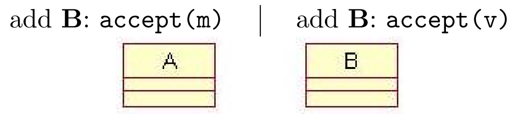


Figure 20: “Add same class” script and outcome in Rational Rose

interesting, the script accepts the left’s idea of the model and the right’s idea of its view. This results in the left model’s class **B** (as determined by its ID) taking the position previously occupied by the right model’s class **B** (Figure 20).

Add Different Classes

Due to RSA’s separation of the model from its view, the second primitive presents a pitfall for the inattentive. It starts with the same ancestor model as the last, but this time adds a different class to each side—**L** on the left, and **R** on the right, as depicted in Figure 21.

Consider what happens if the Merger accepts the entire left contribution and only the right’s model component, then later, for whatever reason (time between actions, UI clutter, etc.), rejects the right’s *view* component. The outcome model will appear to lack class **R** (left half of Figure 22), even though it was actually accepted. An examination of the project explorer tree (right half of Figure 22) will reveal, however, that class **R** is indeed still part of the model. No harm is done, *provided* that any remaining merge actions are predicated on the whole model, as shown in the explorer, rather than on the partial model presented by the view.

It could be argued that a change’s model component should never be accepted without accepting its view component. While probably good advice, it raises the question: Why is the capability to do exactly the converse present at all? Though

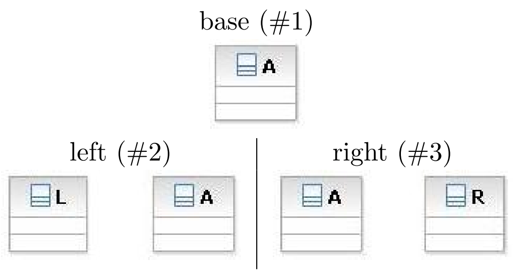
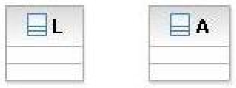


Figure 21: “Add different classes” common base and evolved replica-versions

add **L**: `accept(m,v)`



add **R**: `accept(m), reject(v)`

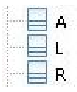


Figure 22: "Add different classes" script and outcome in Rational Software Architect

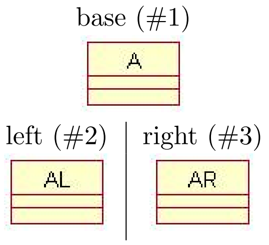


Figure 23: "Rename class" common base and evolved replica-versions

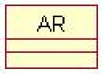
linkage in RSA, from view to model, prevents accepting an element's view component without also accepting its model component, there is no linkage in the opposite direction that would avert this problem. Rose, however, provides linkage in both directions, and thus does not suffer from this particular idiosyncrasy. EA, because it does not impose any distinction between model and view, manages to avoid this issue altogether.

Rename Class

Depending on how a tool handles detection, acceptance and rejection of conflicting changes, even something as simple as renaming a class can catch the Merger unawares. The "rename class" primitive illustrates the potential for more model, view confusion. Starting once again with a single class **A** for the ancestor model, the left Modeler renames the class to **AL**, while the right changes its name to **AR** (Figure 23).

The merge script of Figure 24 directs the Merger to accept the model component of the change from the left, and the view component from the right. Under Rose, the outcome is a class named **AR** (left side of Figure 24). Because this is a rather curious result, a closer look is warranted. Digging into the tree of the project explorer reveals

rename **A** to **AL**: `accept(m)`



rename **A** to **AR**: `accept(v)`

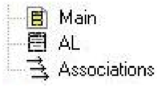
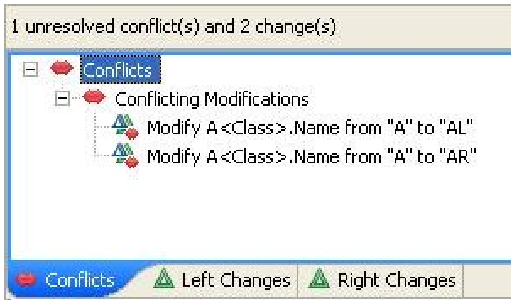


Figure 24: "Rename class" script and outcome in Rational Rose


1 unresolved conflict(s) and 2 change(s)

- Conflicts
  - Conflicting Modifications
    - Modify A<Class>.Name from "A" to "AL"
    - Modify A<Class>.Name from "A" to "AR"

Conflicts | Left Changes | Right Changes

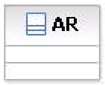
Figure 25: “Rename class” conflicts in Rational Software Architect

that the class is actually named **AL** (right side of the figure). As it turns out, **AR** is merely a label—a class property that Rose gives no apparent way of correcting—and not the element’s name.

We get the same outcome in EA, but for a different reason. EA does not separate the model from the view, and a three-way merge has to be mimicked with two `syncs`. Synchronizing the left model with the base model changes class **A** to **AL**, after which synchronizing the right model with the base model changes the newly renamed **AL** class to **AR**. The `sync` action is not commutative—reversing the order of the actions will result in a class named **AL**.

Two Modelers trying to rename the same class does not cause these tools concern. RSA, however, flags the event as a conflict. For this reason, the two modifications are displayed linked in a mutually exclusive fashion in the *Left Change Pane*, as pictured in Figure 25. The complementary pane shows the same relationship, but from the perspective of the right Modeler, with the placement of the changes reversed. While there is sense in linking conflicting changes in this way, the upper script in Figure 26 shows what happens when the class renaming on the left is rejected—the renaming

rename **A** to **AL**: `reject(m)` | rename **A** to **AR**: `ignore`



rename **A** to **AL**: `accept(m)` | rename **A** to **AR**: `ignore`
rename **A** to **AL**: `reject(m)`

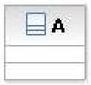


Figure 26: “Rename class” script and outcome in Rational Software Architect

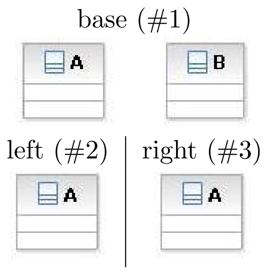


Figure 27: “Delete same class” common base and evolved replica-versions

on the right is automatically accepted. Not only is this “acceptance by rejection” somewhat unexpected, the behavior is not consistent, as explained next.

Observe the lower script of Figure 26. Here, the Merger at first accepts the renaming on the left, which, due to the mutually exclusive linkage of conflicting changes, automatically rejects the renaming on the right. So far so good, but what if, on second thought, the Merger decides to reject the renaming after all? Based on the previous script, one might expect the change on the right to then be accepted by rejection; but as can be seen in the figure, this is not the case. Instead, an “implicit ignore” of the right side occurs, so no class renaming takes place whatsoever.

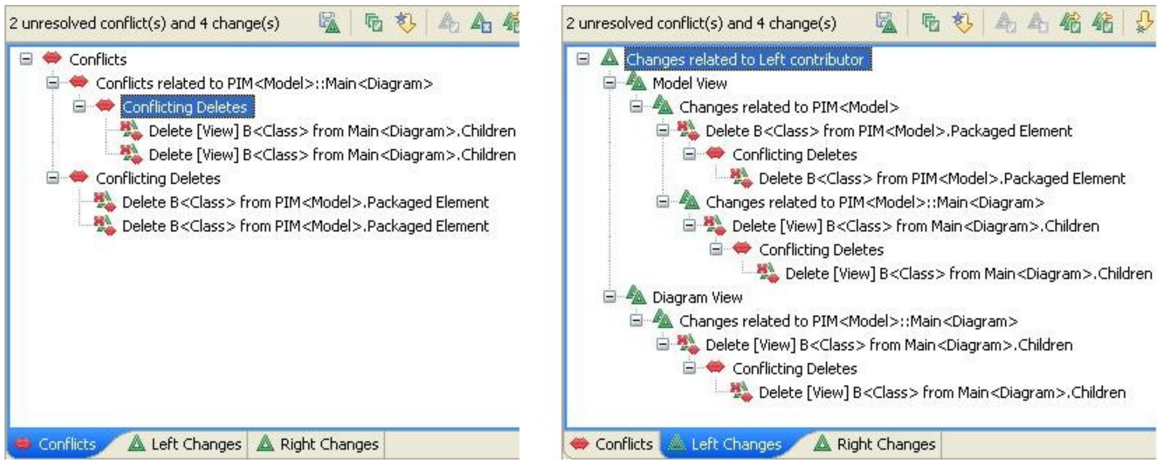
Delete Same Class

The intention of the set-up in Figure 27 is to test the removal of the same class from both sides. The surprise in the case of RSA is not the outcome, which is as expected, but the noise-laden presentation (Figure 28) generated by such a minor modification, *and* that deleting the same class from both sides is considered a conflict.

The tool’s *Left Change Pane* (mirrored by an unshown *Right Change Pane*) portrays the two deletions as linked: If the Merger rejects the left deletion, the one on the right is accepted by rejection, and the class is deleted anyway—another disconcerting example of automatic acceptance winning over explicit rejection. If the Merger accepts the deletion on the left, a more appropriate rejection by acceptance occurs on the right, one that does not offend the intuition.

3.3.2 Textual Tools

Comparable Java representations of the models in the merge scripts of the primitive evaluation cases that are executed under CVS yield the following observations (similar



2 unresolved conflict(s) and 4 change(s)

- Conflicts
  - Conflicts related to PIM<Model>::Main<Diagram>
    - Conflicting Deletes
      - Delete [View] B<Class> from Main<Diagram>.Children
      - Delete [View] B<Class> from Main<Diagram>.Children
  - Conflicting Deletes
    - Delete B<Class> from PIM<Model>.Packaged Element
    - Delete B<Class> from PIM<Model>.Packaged Element

Conflicts | Left Changes | Right Changes

2 unresolved conflict(s) and 4 change(s)

- Changes related to Left contributor
  - Model View
    - Changes related to PIM<Model>
      - Delete B<Class> from PIM<Model>.Packaged Element
        - Conflicting Deletes
          - Delete B<Class> from PIM<Model>.Packaged Element
      - Changes related to PIM<Model>::Main<Diagram>
        - Delete [View] B<Class> from Main<Diagram>.Children
          - Conflicting Deletes
            - Delete [View] B<Class> from Main<Diagram>.Children
  - Diagram View
    - Changes related to PIM<Model>::Main<Diagram>
      - Delete [View] B<Class> from Main<Diagram>.Children
        - Conflicting Deletes
          - Delete [View] B<Class> from Main<Diagram>.Children

Conflicts | Left Changes | Right Changes

Figure 28: "Delete same class" conflict and left change panes in Software Architect

results are obtained under Subversion):

1. *Add same class*: Files for class **A** and class **B** coexist in the repository; any attempt to add a second class **B** causes a conflict.

2. *Add different classes*: Files for classes **A**, **L** and **R** coexist in the repository; no conflicts are raised.

3. *Rename class*: Files for classes **AL** and **AR** coexist in the repository; class **A** is shown as having been removed from the project; no conflicts are raised.

4. *Delete same class*: The deletion of the file for class **B** is propagated to other workspaces upon update; no conflicts are raised.

All of the favorable outcomes rest to some degree upon the fact that Java forces file names to be the same as the classes they define. The results for C++ would be quite different if the same convention were not followed. Be that as it may, the file and text-based nature of both configuration management systems, along with our awareness of that nature, dispose us to accept these results as perfectly reasonable. In general, practitioners are satisfied with the state of textual merging. So much so that measurements of textual merging in an industrial setting place the success rate of automatic merging at around 90% [Perry et al., 2001].

3.3.3 Research Alternatives

Several groups are pursuing model management issues that touch on model merging. Comparing vendor tool outcomes to those obtained by running the same evaluation cases in research tools has proved instructive. Out of the box, Epsilon was the only alternative considered that fared better than the vendor tools, so it is the only one covered here.

Epsilon is relatively mature, easily available, and offers solid documentation and an active newsgroup. It is now a component of the Eclipse Generative Modeling Technologies (GMT) project [Kolovos et al., 2006b]. It provides a collection of metamodel neutral, task-specific, model management languages. Of most interest here are its rule-based comparison and merging languages (ECL and EML), and by extension, the Epsilon object language (EOL).

Model comparison with the ECL produces a file that consists of two models: one containing all elements that were matched between the contributions with the matched elements melded into a single element, and the other containing any remaining unmatched elements. The EML is constrained in ways similar to EA: merging proceeds automatically with no user input, and there is no three-way merge capability. This latter constraint means it cannot double check changes against a common base, as required for the "rename class" and "delete same class" cases.

3.4 Expectations and Counterintuitive Behavior

Until such time as a formal definition of model merging becomes widely accepted (e.g., [Brunet et al., 2006], [Westfechtel, 2010]), precisely how a model merging tool should behave will largely be determined by developer expectations. These expectations have, to a large extent, been shaped over many years by the merge capabilities of textual *diff*-like tools and version control systems. Developers thus, have been steadily instilled with an "intuitive" sense for how merging should proceed.

It is from a developer's intuitive point of view that Table 2 offers a qualitative comparison of each tool's performance on the primitive evaluation cases with regard to merge effectiveness (*merge*), and presentation and control of the merge (*usage*). Admittedly, the sample size is small, and the results are somewhat subjective, being

Table 2: Qualitative summary of tool outcomes for evaluation cases

	Add Same		**Add Diff**		**Rename**		**Del Same**	
	merge	usage	merge	usage	merge	usage	merge	usage
RSA	U	U	S	U	S	U	S	U
Rose	S	U	S	U	U	U	S	U
EA	U	n/a	S	n/a	U	n/a	S	n/a
CVS	S	S	S	S	U	S	S	S
SVN	S	S	S	S	U	S	S	S
Epsilon	S	n/a	S	n/a	n/a*	n/a	n/a*	n/a

S=satisfactory, U=unsatisfactory, *=requires 3-way merging

dependent on evaluator expectations. Still, some broad conclusions may be drawn:

- If the merge results obtained are not largely unsatisfactory, then usually the interactions with the tool are.
- Proper matching is required for satisfactory merge outcomes. Failure to do so is at the root of many failures. Matching solely on ID is not sufficient.
- The performance of textual merging is misleading—source code is not a high-level model—and, in the case of Java, accidental, with success being attributable to the direct mapping between class and file names.
- Epsilon succeeds where the others fail, because it has more sophisticated matching capabilities and can take some semantics into account.
- The tools considered, perform state-based merging. The conclusion regarding other approaches is that most tools do not attempt to use them.

Routine, poor execution of a task as central to cooperative MDE as model merging is, does little to convince practitioners of the value of models, or management of the productivity gains intrinsic to model-driven engineering. Better model merging tools are essential. The recommendations for improving their capabilities and design that follow are made in this light, and the guidance proffered by them has been used in the advancement of this work.

3.5 Recommendations for Model Merging Tools

Before proceeding with recommendations it must be noted that experimentation, and not documentation, is the basis for many of our generalizations regarding the behavior of the evaluated tools. This is a situation that is good for neither the investigator nor the user. Vendors should strive—and this may be taken as the first recommendation—to make the logic that drives their tools explicit and, ideally, accessible to inspection and even open to modification. This is particularly important for the tasks of model element matching and conflict resolution.

Improve Model Element Matching

More effort is needed to establish correspondence between model elements. Element ID works well for matching elements in the base model to elements in its evolved replicas, but not across replica-versions. When a pair of IDs are identical, it is almost always safe to consider the elements as matched. However, when they differ, nothing at all can be said about the elements as a pair without taking other criteria into consideration.

Element identity confusion is the root cause of RSA and Rose producing opposing results for the "add same class" case. RSA uses only identifiers to match elements and, hence, misses that the **B** classes on both sides of the merge may be the same class. Not wanting to miss any changes, the merged outcome ends up with two classes with the same name. Rose recognizes that the Modelers have both added a class **B** and matches them, but then silently and somewhat arbitrarily takes up one and discards the other. It thus avoiding duplicate elements, at the risk of throwing away a valid change.

Both outcomes may be correct in different circumstances. Though the Rose merge is apt to be more right than wrong, it will be more difficult to fix than the RSA outcome if the match is incorrect. The deciding factor should be modeler intent. If this cannot be discerned, the tool should make it possible to review and alter the matching, or perhaps even rename one of the classes. As it stands, if the Merger does want to do a renaming, the only option is to abort the merge—a steep price to pay if a great deal of time has been spent merging prior to reaching this point.

In general, overreliance on element identifiers is pervasive and is the source of many difficulties encountered by model merging tools. The match rules of the Epsilon Comparison Language shine here, since they are able to bring semantics to bear on the problem. Unfortunately, this solution is not perfect, because it entails either finding a pre-written set of rules for the models or model domain at hand, or writing one's own set.

Widen Conflict Detection

Trivial as it is, none of the modeling tools detected a conflict in the "add same class" evaluation case when one should have been reported. EA, which only tracks database records of model elements and not structure, does not detect most conflicts, so its outcome is to be expected. RSA and Rose, however, do consider structure to some extent, and yet neither flags two model elements of the same name as worthy of scrutiny. CVS and Subversion, on the other hand, do force the Merger to deal with the situation by signaling a conflict.

With a little insight into their workings, a logical explanation for each tool's failings can usually be found. Still, undesirable behavior is undesirable, even if a reason is at hand. The question remains: Should a developer have to base expectations of model merging on how a particular tool is implemented? This particular pair of changes should be flagged as a conflict: either because they are adding the same element, in which case the classes should be melded into one, with an appropriate assumption of properties and dependencies; or because keeping both classes will create an invalid model, in which case the opportunity to perform a rename should be made available.

Enhance Conflict Resolution

Besides invalidating a model, careless or incorrect conflict resolution can wipe out entire contributions, as when Rose discards one class in the "add same class" evaluation case. To avoid doing likewise, and since its global synchronization rules offer no insight into model structure, EA necessarily follows a conservative strategy of accumulation in which its outcome is the union of its two input models, less any deletions made by either Modeler. EA thus has little need to detect conflicts, except in those cases where one Modeler is deleting an element the other has modified.

However, even tools that *do* detect conflicts and which *do* consider model structure follow a similar strategy when it comes to deletion. They generally make the *assumption of the competent Modeler*. In other words, if a Modeler deletes an element that is in both models, there is probably a good reason for this; and further, that the model will be better off for it, and the other Modeler will not object. Still, it is an assumption, and one that most tools make no effort to verify.

When a conflict *is* raised, the choice of possible actions in most merge tools is limited to accepting or rejecting one of the changes *in total*. That is, there is no way to retain or discard only a portion of a change. Imagine a tool that considers the addition of two classes with the same name to be a conflict, and that the Merger accepts one of the changes (implicitly rejecting the other). The tool should provide a way to transfer remnants of the rejected class to the accepted class. This comes down to issues of change granularity and element containment.

Lastly, for models of any real size, meaningful resolution mandates three-way merging. The extra insight gained from the base model eliminates ambiguity and overly conservative strategies by enabling a tool to pick changes from the correct contribution with greater precision [Letkeman, 2005a]. Of the evaluated tools, only EA was unable to perform a three-way merge.

Simplify Conflict Presentation

Rational Software Architect and Rational Rose both have sophisticated graphical interfaces to solicit feedback from the Merger during a merge session. Enterprise Architect lacks such a GUI, because it only supports synchronization, but it does provide a dialog for undoing individual actions after the fact.

Model merging is a complex undertaking requiring well-informed user input; consequently, tool presentations of changes tend to be very information-rich. However, a screen cluttered with model, conflict and view data can generate enough cognitive noise to overwhelm comprehension. Tools need to make judicious use of all this information, preferably presenting it over multiple levels. A big picture is invaluable for grasping the change context, while change-element relationship details are vital for appreciating the impact of a change's acceptance or rejection. The Merger should be able to move between these levels at will, without being inundated with graphics.

Redundancy needs to be ruthlessly slashed, while keeping related data visually close.

To advance user understanding of the merge, and to enable more prescient merge decisions on their part, it should be possible to trace out and determine just what merge actions run against which contributions have gone into creating the merged model in progress. Additionally, the user should be able to view the information relevant to any one conflict in isolation—one merge decision at a time. This kind of conflict management interface could also form the basis for a merge undo facility—a feature that would enable exploratory merging.

As a counter example to the above, in the “add same class” case, Rose produces a default outcome with no input from the Merger. In order to determine from which model the accepted change came, one has to manually find and match element IDs. Since the tool already knows which elements go with these identifiers and what action have been taken in regard to the elements, it could make this connection explicit with little effort.

Graphical linking is one way to emphasize the relationship between related changes while reducing their visual distance. RSA does this quite well, and almost gets it right. Its linkage toggles the related changes from the two sides between acceptance and rejection in a mutually exclusive fashion. For acceptance, this is exactly what a conflict demands. Only one change can become part of the outcome model. The other must be rejected—what we have termed “rejection by acceptance.” The converse, “acceptance by rejection,” is problematic. It makes it possible for a change that is explicitly rejected to end up in the merged model anyway, as happens in the “rename class” evaluation case.

The same case produces an “implicit ignore” in RSA. Accepting a change on one side rejects its conflicting counterpart on the other. Subsequently rejecting the same change will *not* toggle the automatically rejected change back to accepted. Instead, both changes will be rejected (i.e., ignored). This runs counter to expectations set up by the “acceptance by rejection” behavior observed in other cases. Here, our prior experience is contradicted, which is probably as good an example of counterintuitive behavior as one could want.

Separate Model Issues from View Issues

Closely related to presentation issues is the tying of the model to its logical view(s). While maintaining such ties during modeling in general may be useful, its benefit during model merging is questionable, as the "add different classes" test makes clear. In this test, the merge hides one of the added classes due to the interplay of merge actions—one modeling operation (add a new class) requires two merge actions (accept the model change *and* the view change).

A modeling tool rightfully needs to be concerned with views, but its merging should concentrate, first and foremost, on the models, not on its graphical representations of those models. Even for small models, the mental bookkeeping required to shift through the sand of view details for relevant modeling nuggets can be boggling. An argument can be made for being able to move between a model and its view during merging, but this should be at the discretion of the Merger, not the tool. A tool that updates its views based on a merge should perform merging in two steps: first completely merge the models and only then the models' views, using whatever information has been gathered in the first merge. This way, there will be no need for the user to consider both perspectives at the same time.

By the same token, the model should be the source of all human-readable names, not the view. Recall, in the "rename class" case under Rose, how the left class became labeled with the name of the right class, because the tool gave precedence to a view *label* in its conflict display, over the model element's actual *name*. The fact that the class that ended up in the model was really from the left model was obscured.

Perform Post-Merge Model Validation

Tools should do a cursory validation of the model after merging and *before* the outcome is saved. Even better, if at all possible, the model should be validated at each merge decision. Current integrated MDE toolsets take a decidedly non-integrated approach when it comes to merging. While most tools have verification capability, their stance seems to be that model validation can wait until code generation. But inconsistencies and conflicts need to be uncovered when they originate—when the facts that gave rise to them are fresh in mind.

For instance, a quick check immediately after the "add same class" case would have alerted the Merger that their model now had two classes named **B**. If the duplication were revealed at that time, a decision as to which of the contributions to keep, eliminate, or rename would be easy to come by.

Chapter 4

The Merge Space

That a model change operation can be thought of as a subtransformation of a larger model transformation was recognized in the background (Section 2.9). If the transformation is endogenous, then when used to patch a model M, conforming to metamodel MM, the transformation will "grow" M, one subtransformation at a time, into a new version of itself M', that also conforms to MM. Only endogenous transformations are considered in this work.

Building on the notion of change operation as subtransformation, this chapter introduces a conceptual space in which the merging of transformations can occur. A second device used for tabulating the dependencies and conflicts that can arise between the subtransformations of that space is also presented.

4.1 Model Operations and the Change Plane

In Figure 29, we orient the transformation grid of Lippe and van Oosterom [1992] to resemble the classic merge diamond (Figure 10), with nodes representing models and edges change operations, or subtransformations. The changes along the axes comprise the transformations T_L and T_R, which transform the base model M_B into left and right versions of itself. To follow a path from M_B to another node is to execute the sequence of changes that lie on the path, transforming M_B at each step. Hence, each node hosts the set of models produced by all paths that reach it. We term this surface of subtransformations and potential models the *change plane*.

Interactive merge tools provide for implicit control over what transpires on the

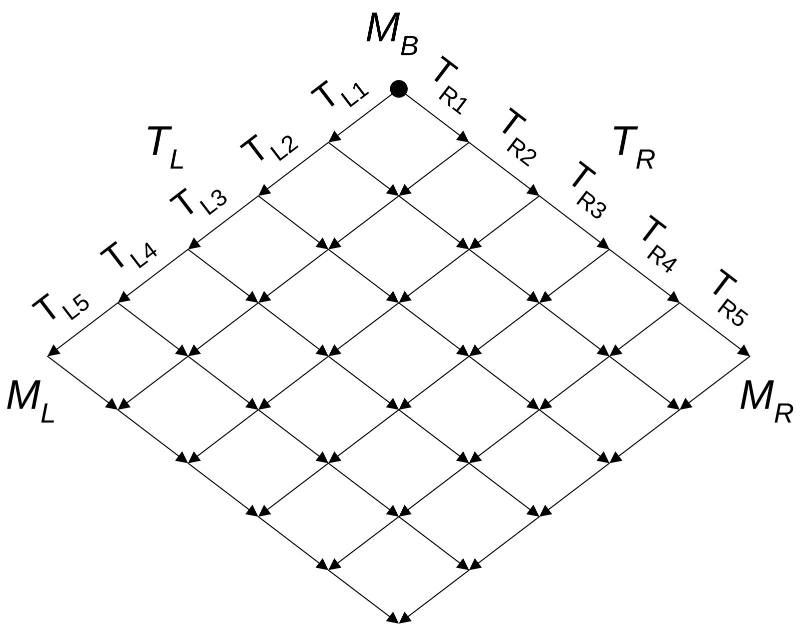


Figure 29: Change operations projected as subtransformations onto change plane

change-plane when they seek input during a merge session, whereby the user accepts or rejects model changes with the intent of merging changes. From a transformational point of view, to move forward (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. To move backward—though disallowed for constructing a weave—is to *undo* an edge's change.

Rejecting a change poses a problem if, in order to maintain rigor, we stipulate that all movement between nodes must take place along connected edges. In Figure 30*a*, the Merger has rejected T_{L_3}, T_{R_3}, and T_{R_4}, essentially ripping a gash across the plane that the merge process must somehow "leap." Reordering the changes, as in Figure 30*b*, or simply discarding the rejected ones, mends the plane, thereby satisfying the connected edge movement stipulation.

4.1.1 Conflict-Free Region and the Frontier Set

All weaves for a particular node consist of the same operations and, hence, differ only in their ordering. Every node on the plane hosts the set of models produced by all the weaves that terminate at that point (disallowing backward travel). A

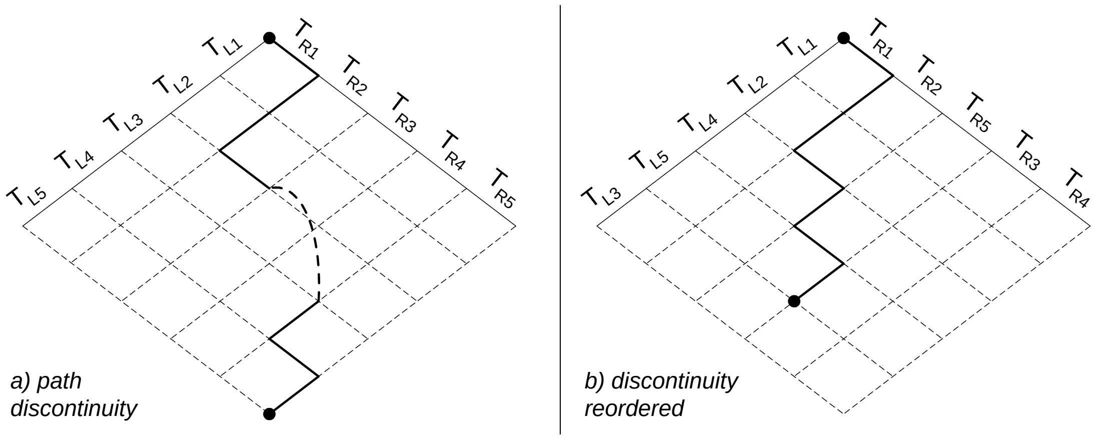


Figure 30: Mending change-plane discontinuities induced by merge decisions

node that hosts the same model for every one of its weaves (i.e., all of the involved operations commute) is conflict-free and said to be *single-valued.* In contrast, a node that hosts more than one model (i.e., at least one pair of the involved operations does *not* commute) is in conflict and said to be *multiple-valued.* The first conflict encountered will lay out a border of multiple-valued nodes across the change plane called the *frontier set* [Lippe and van Oosterom, 1992]. This is the boundary at which conflict-free subtransformations can no longer be acquired, and heuristics or user action will be called for. Clearly, the further the frontier set can be pushed out from the origin, the fewer decisions the user will need to be confronted with.

The example in Figure 31 serves to clarify. Consider a simple class model consisting of classes **A** and **B**. In part *a* of the figure, the base model is modified by the left transformation to initialize a member of **A** and **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 by which any one of them can be reached (disallowing backward travel).

Since **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 this node multiple-valued. The other nodes are multiple-valued for the same reason. The two nodes along the bottom right border host additional models due to another conflict—modifying **B**.y and deleting class **B** do not commute. For this structuring of transformations, the frontier set consists of those multiple-valued nodes in Figure 31*a* marked with an ‘F’. Some of the paths going to the outermost node also suffer from an inconsistency, because

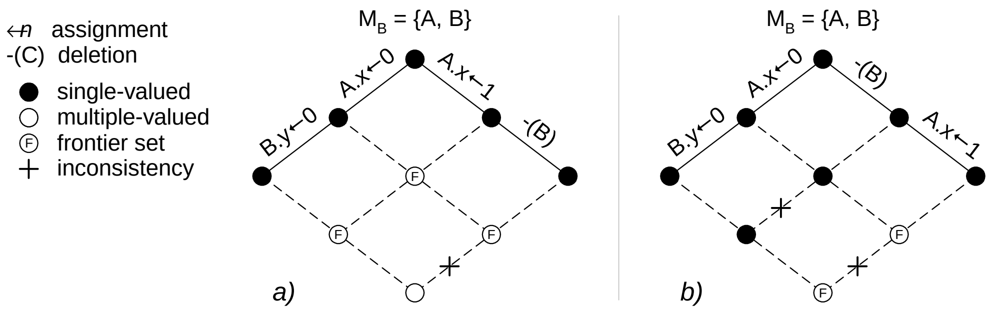


Figure 31: Using operation commutation to push the frontier set outward

attempting to access **B** after it has been deleted violates a precondition of assignment, namely, that the target exist.

Swapping the positions of the right-hand subtransformations, as is done in part *b* of the figure, delays confronting the conflict, turning two nodes into single-valued points. The frontier set has thus been expanded slightly: whereas the first structuring could only incorporate two changes that came from the same side ($\{\mathbf{A}.\mathrm{x} \leftarrow 0,\ \mathbf{B}.\mathrm{y} \leftarrow 0\}$, or $\{\mathbf{A}.\mathrm{x} \leftarrow 1, -(\mathbf{B})\}$), the second can incorporate one entire side and half of the other ($\{\mathbf{A}.\mathrm{x} \leftarrow 0,\ \mathbf{B}.\mathrm{y} \leftarrow 0, -(\mathbf{B})\}$). This comes at the expense of another inconsistency, which is ordinarily of negligible cost, since operation commutativity can usually be taken advantage of to find another path in which the circumstance that causes the inconsistency does not occur.

4.1.2 Direct and Indirect Conflicts

The simple merge illustrated in Figure 32 will serve as an example for the remainder of the chapter. The figure's change plane has been populated with the operations from the lists appearing on either side of the plane. The operations make up the subtransformations of the two transformations T_L and T_R, which in turn transform replicas of base model M_B into the left and right versions M_L and M_R, respectively.

The nodes drawn as solid black dots are single-valued—the same model is produced by whatever path is followed to reach them and, hence, the operations that lie on these paths are without conflict. The rest are multiple-valued—different models are produced by at least two of the paths the lead to them, and therefore contain operations that are in conflict. But not all conflicts are created equal, prompting us to make a distinction between direct and indirect multiple-valued types.

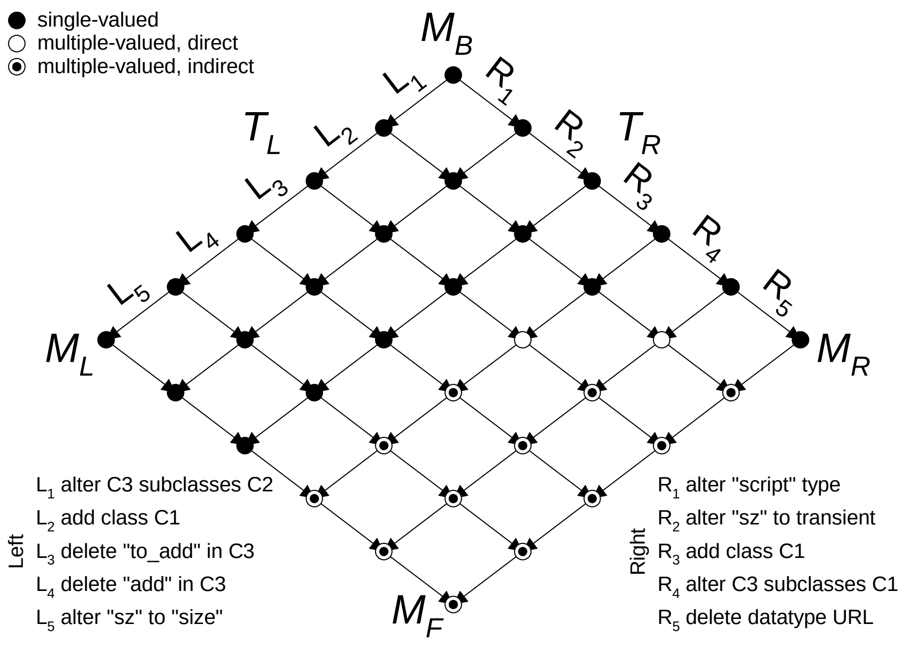

● single-valued
○ multiple-valued, direct
◉ multiple-valued, indirect
M_B
M_L
M_R
M_F
T_L
T_R
L_1
L_2
L_3
L_4
L_5
R_1
R_2
R_3
R_4
R_5
Left
L_1 alter C3 subclasses C2
L_2 add class C1
L_3 delete "to_add" in C3
L_4 delete "add" in C3
L_5 alter "sz" to "size"
Right
R_1 alter "script" type
R_2 alter "sz" to transient
R_3 add class C1
R_4 alter C3 subclasses C1
R_5 delete datatype URL

Figure 32: Change plane overlaid with subtransformations of left and right versions

If the subtransformations that *terminate* at a node do not commute, we say that the node is a direct multiple-valued type and it has a *direct conflict.* When the terminating subtransformations at a node do commute, but at least one pair of subtransformations in the path taken to reach the node do not commute, the node is an indirect multiple-valued type and it has an *indirect conflict.* Suppose model matching determines that changes L_2 and R_3 create the *same* element. If the added classes are not identical, the operations will not commute and will be in direct conflict. Then the L_3, R_3 “downstream” node, in spite of the fact that its terminating operations *do* commute, will be (indirectly) multiple-valued.

In fact, every node reached by a path that passes through a direct conflict node will suffer from its conflict. This spillover is what produces the frontier set—the upper tier of multiple-valued nodes running across the plane. A direct conflict is analogous to that of the keystone in an arch: pull the keystone, and the other stones fall; resolve a direct conflict, and its indirect ones vanish. This analogy tells us that, instead of resolving all 14 conflicts that lie in the plane, we need only resolve the two *key conflicts.*

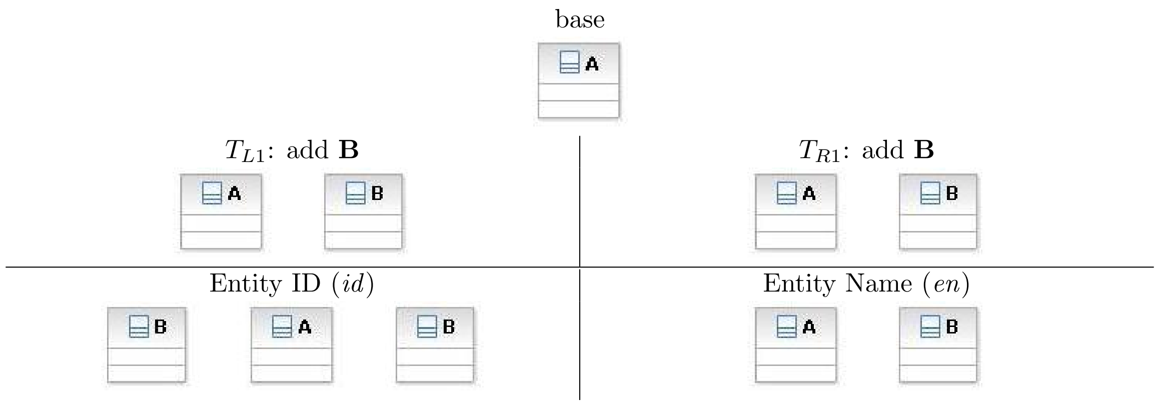


Figure 33: Model element matching strategies for “add same class” example

4.2 The Matching Dimension

This section takes a closer look at the issue of model element matching across the replica-versions of a merge session. The “add same class” primitive evaluation case of Subsection 3.3.1 is used to explore matching in more detail. The set up for the evaluation case in which a new class **B** is added to both sides, is repeated in the upper portion of Figure 33. As already seen, Rational Software Architect will decide that the classes are different and thus succumb to the duplicate element problem, producing a model with two **B** classes (lower right), while Rational Rose will draw the opposite conclusion and combine the new classes into one (lower left).

The incongruity of the results stems from the different ways in which the tools match the elements of one model to those in the other, i.e., their differing matching strategy. Two strategies are contrasted: one based on entity ID and the other on entity name. Other strategies are possible, for instance, ones based on structural similarity or on model dependencies. Most tools, however, support only one strategy no matter the circumstances, a shortcoming that will concern us later.

Matching strategies of a merge session can be visualized as representing a third dimension, orthogonal to the change plane, with each strategy occupying, in effect, its own change plane. The conflicts created by the weaves on a particular plane are those that result due to that plane’s matching strategy. The family of parallel planes thus formed (Figure 34) defines a three-dimensional solution space, or *merge space*, of strategies, subtransformations, and transformed models that is applicable to the models of the given merge session.

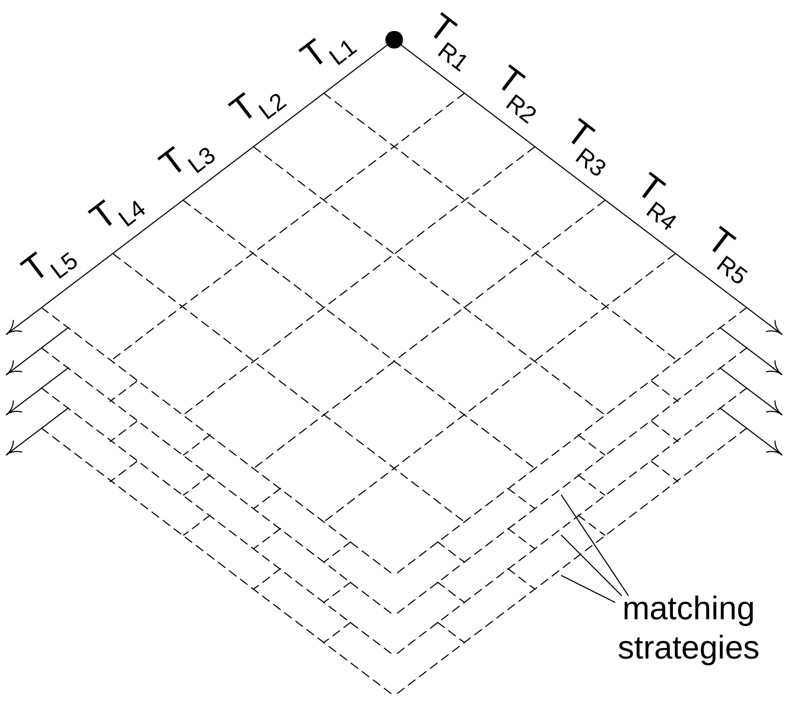


Figure 34: Matching strategies spread across a family of change planes

4.2.1 Matching Strategy Conflicts

Ideally, a merge should not be confined to one matching strategy, but instead be free to use the strategy that gives the best measure of similarity for each element pairing. Thus, a merge should be able to move in the third dimension of the merge space to search for the most appropriate strategy for the model elements under consideration. However, this freedom comes at the cost of potential conflicts across strategy planes.

Consider again the example of Figure 33, in which both Modelers add a class **B** to their respective replicas. The Merger can accept one, both, or neither of these changes. These merge choices yield five possible outcomes of the merge for *each* strategy. The permutations of choices are enumerated in the first column of Figure 35, and the resulting outcomes for two matching strategies in the adjacent columns, where strategy *id* matches by entity ID, and strategy *en* by entity name.

Taken in isolation, all the nodes of both strategy-specific change planes are single-valued: the models produced at every node are the same irrespective of the path traveled to reach that node. However, if we look *across* the two planes, the node reached by the fourth and fifth weaves is multiple-valued. In Figure 36, a miniature

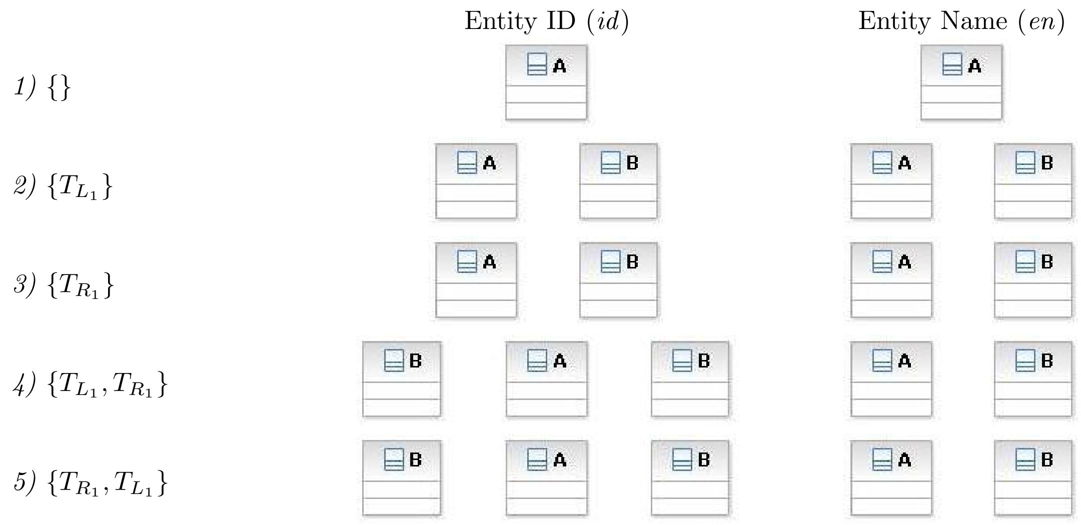


Figure 35: Merge decision permutations for “add same class” example

representation of the outcome produced by each weave has been placed alongside the terminating node for the weave and adorned with the number(s) of the permutation from Figure 35 that produced it.

Reasoning by analogy, we can 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. To “resolve” such strategy conflicts, a merge tool that offers multiple matching strategies must come up with some criteria, or method, by which the best strategy can be selected for any given pair of model elements.

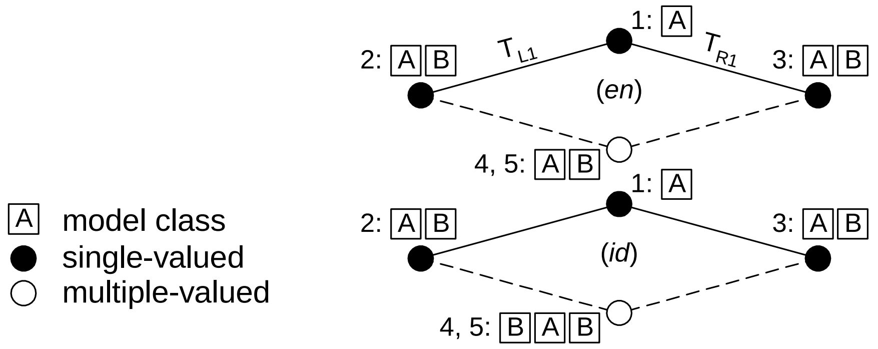


Figure 36: Cross matching strategy conflict for “add same class” example

			L_1	L_2	L_3	L_4	L_5	R_1	R_2	R_3	R_4	R_5
Left	L_1 alter C3 subclasses C2	L_1									$\times$	
Left	L_2 add class C1	L_2								$\times$	$<$	
Left	L_3 delete "to_add" in C3	L_3				$<$						
Left	L_4 delete "add()" in C3	L_4			$\wedge$							
Left	L_5 alter "sz" to "size"	L_5										
Right	R_1 alter "script" type	R_1										
Right	R_2 alter "sz" to transient	R_2										
Right	R_3 add class C1	R_3		$\times$							$<$	
Right	R_4 alter C3 subclasses C1	R_4	$\times$	$\wedge$						$\wedge$		
Right	R_5 delete datatype URL	R_5										

Figure 37: Conflict matrix after application of *before* predicate to all change pairs

4.3 Change Ordering and Partitioning

The change plane uncovers conflicts, but not the dependencies needed to order the changes. The techniques of this section are intended to establish the order of the change operations in the merged operational trace (i.e., merged transformation).

4.3.1 Visualizing Relations with the Conflict Matrix

A tabular layout of the relations between change operations is helpful for settling change execution order. Applying the *before* predicate of Section 2.9 to the changes of the change plane in Figure 32 yields the resulting *conflict matrix* of Figure 37 (which also reproduces the changes). A $<$ or $\wedge$ points to which of a matrix cell's operations has precedence, or positional preference (explained below), over the other, as determined by the *before* predicate. It is easiest to read against the wedges, so going across the third row reads, "L_3 is preferred to L_4," as does going down the third column. The notation makes the symmetry along the diagonal of the matrix evident. A careful comparison reveals that the conflicts of the matrix (marked with a '$\times$') coincide with the direct conflicts of the change plane.

Quadrants II and IV depict same-side changes, and quadrants I and III, cross-side ones. Since same-side changes of a consistent model cannot be in conflict, there is no ambiguity over what *before* means in quadrants II and IV: An operation must come before another when required to satisfy preconditions. For example, change R_4 will fail for lack of the targeted superclass, if R_3, which creates the class, has not yet been executed.

Across models, the meaning is not so clear. For instance, if we assume single inheritance, changes L_1 and R_4, which attempt to give class **C3** different superclasses, obviously clash. However, there is no *a priori* reason for L_1 to come before R_4, or R_4 before L_1, and, since this is an *overwrite* situation, every reason for it to come *after*. Thus, if interpreted strictly in terms of precedence, *before* will not detect a conflict in this case. Therefore, it is more proper to think of the predicate as establishing *positional preference*, rather than precedence, and to write the cross-side version of *before* to take a selfish "me first" (or "last," as the case may be) attitude.

To relate the theoretical discussion here to the pragmatics of our prototype, we take a small detour to explain the operation of *before* in terms of the decision table rules to be introduced in Subsection 7.4.3. First, consider the L_3 and L_4 pair of same-side changes. Change L_3 deletes the parameter `to_add` from operation `add`, which is in turn deleted by L_4. This is a delete-delete change-pair in which one of the elements, the parameter, is contained by the other, the operation. The pattern matches rule 3 of the `before_same` decision table (Figure 63), which stipulates that the inner delete operation (L_3) must be done before the outer one (L_4).

Next consider the cross-side changes L_1 and R_4, which try to give class **C3** different superclasses. In this case, rule 7 of table `before_cross` (Figure 64) fires, no matter the order of the operands. The rule claims priority for L_1 in one direction of comparison, and for R_4 in the reverse, thus flagging a conflict between the two changes. The other cross-side pair, L_2 and R_3, are not in conflict, *unless*, they are deemed by element matching to add the *same* **C1** class (albeit with different IDs) to their respective sides, in which case, rule 1 will find them in conflict. If no conflict were raised after such a match, the model would end up with two classes of the same name—the duplicate element problem.

Some tools might consider L_5 and R_2 to be in conflict, but a condition of table `before_cross`, called `alter_same_property`, recognizes that the operations actually alter *different* properties of the same element and therefore do not conflict. If no decision table rules are asserted for a given change-pair, there are no preconditions to satisfy and the operations may be performed in any order.

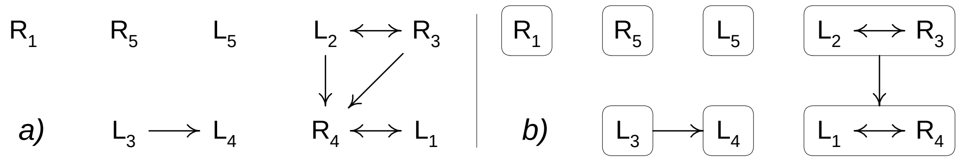


Figure 38: *a)* Change dependencies, *b)* partitioned into mutually exclusive blocks

4.3.2 Breaking Cycles with Conflict Partitioning

The conflict matrix is essentially a graph of change dependencies (e.g., Figure 38*a*). Determining the execution order of change operations directly from the graph can be difficult if conflicts are present or, if cycles are involved, impossible. To remedy this, *conflict partitioning* gathers conflicting changes into collectively exhaustive and mutually exclusive blocks.

This is accomplished by: 1) putting non-conflicting operations into their own *block*, 2) putting each pair of conflicting operations into a block, with a double-headed arrow between them, and 3) adding single-headed arrows between blocks to reflect the ordering of the contained changes of each block as a whole, as dictated by the conflict matrix. Following this procedure for the graph in Figure 38*a* results in the partitioning shown in part *b*.

Dependency cycles, like the one in Figure 39*a*, require some additional steps. Partitioning by conflict produces the blocks of Figure 39*b*. The arrows emanating from block L_1, R_1 signify that the other blocks are dependent on the outcome of that conflict—they require either L_1 or R_1 as a target for their operations. The transitive R_1–R_3–R_2 relation makes the L_1–L_2 relation superfluous, so it is discarded.

Though exhaustive, this partitioning is not mutually exclusive (L_2 is in two

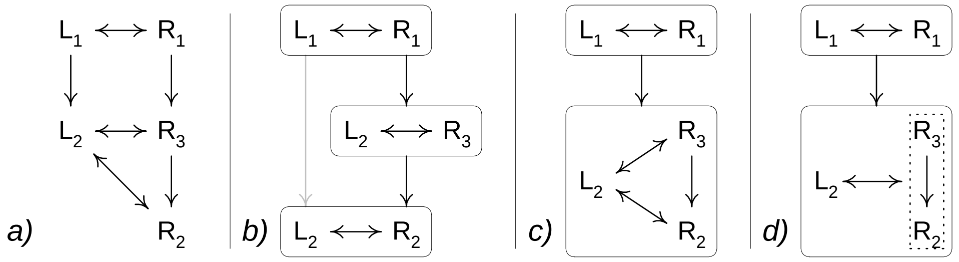


Figure 39: *a)* Change dependencies, *b)* non-exclusive conflicts, *c)* exclusive conflicts using a composite block, *d)* with conflicts collapsed

blocks). Putting the changes of the troubled blocks into a single *composite block* (Figure 39*c*) fixes the non-exclusivity at the expense of reconstructing the cycle. This can be broken by *collapsing* the conflicts between individual operations of the cycle into a single conflict between same-side *dependency chains*. In Figure 39*d*, the R_3–R_2 chain is now treated as a single operation—the two individual conflicts with L_2 have been collapsed into one conflict with the chain as a whole. Collapsing conflicts lowers merge complexity at a slight loss in resolution choices: the closed nature of dependency chains precludes selecting certain combinations of operations. Usually, however, the choices lost are illegal (e.g., accepting L_2 over R_3, *and* R_2 over L_2), and will not be missed.

The merge workflow to be described in the next chapter makes heavy use of the duality of changes and transformations to move freely between state-based and operation-based techniques. It also conducts its merging in the merge space, as weaves of subtransformations, using the *before* predicate to detect conflicts. And the change operations that eventually go into making up the final merged transformation are ordered using the technique of conflict partitioning.

Chapter 5

The Hybrid Merge Workflow

The last chapter provided the means and techniques for visualizing and carrying out a merge in a space of transformations and matching strategies. These concepts are general purpose in that they apply regardless of the type of the artifacts being merged and the merge approach being taken. This is true because any merge has to determine what changes have been made to its artifacts, and because any change can be abstracted into a transformation on those artifacts.

This observation leads to a fundamental insight: *if one has state, one can get operations*, and *if one has operations, one can get state.* To be sure, there are still differences between the two, and they are not inverses of each other—the state-based approach has less information available to it, and can be seen as a simplification of the operation-based approach. Or, if looked at from the opposite direction, as being "enhanced" by the operation-based approach [Koegel et al., 2010].

With this understanding, we propose the solution foreshadowed by the thesis statement; namely, that the problem of model merging can be best addressed by taking a tack that employs the most appropriate merge approach for the task at hand—creating, in effect, a hybrid workflow for model merging. The proposed workflow is premised on the above insight relating artifact state to change operations, which must, however, be qualified: change operations derived from state are neither complete nor ordered, and state obtained from change operations may be incomplete or missing. Both qualifications will be treated after first giving an overview of the approach in the next section. Screenshots of the prototype using the workflow are to be found in Chapter 7.

5.1 Workflow Overview

The proposed solution starts out conducting preparatory work in a state-based mode, transitions to operation-based activity for the merge itself, and returns to state-based techniques to produce the final merged model. The workflow is broken into eight phases. This section outlines each phase in brief, with more detailed descriptions to follow. Mention of the technologies employed and any implementation features have been avoided. Details for the most important of these are given in Chapter 7, and implementation notes for each phase in Appendix A.

The phases of the workflow have been diagrammed in Figure 40 using a pipe and filter architecture style, augmented with control interfaces to allow filters to receive user input [Shaw and Garlan, 1996]. The transformation function for a given filter is explained in the detailed description of that filter's phase. The ordering of the phases is not completely determined—the Model Comparison and Model Differencing phases may be swapped, and indeed are in the prototype—but here, to facilitate understanding, the order involving the fewest complications has been chosen.

The overview is written as if the user were playing the role of Merger and actively engaged at two points in the process. This is the case for the *interactive mode* of operation, but another, in which there is no involvement, called *batch*, or *script*, mode, is also available. In batch mode, all information required by the merge session comes from the command line, a configuration file, or a combination of both. Both modes produce a merged model, but since no manual conflict resolution occurs in batch mode, its model is more likely to be partial. In addition, both modes generate an in-depth report of how the session progressed.

Model Normalization

Input to the workflow is three models (two for a two-way merge). The artifacts are normalized to conform to a common metamodel. This *normal-form* is state-based, regardless of the original type(s) of the artifacts. The internal working models are thus *decoupled* from their originating MDE tool or change operation recording unit.

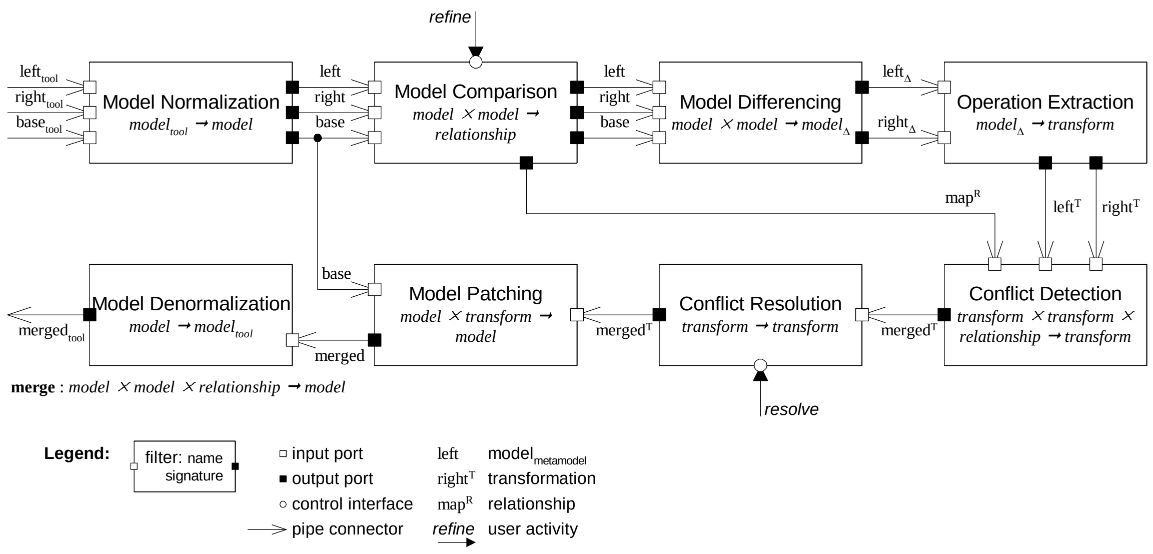


Figure 40: Phases of the hybrid merge workflow as a modified pipe and filter system

Model Comparison

This phase establishes *element correspondence* between the state-based normal-forms of the left and right models. The models are compared so as to match, or map, the elements in one model to those in the other. Matched, or paired, elements are considered to be the same element, and their changes must be assessed for conflict.

Model Differencing

A state-based difference is taken between the normal-forms of the common base model, and its left and right replica-versions. The two resulting *difference*, or *delta*, models conform to the normal-form extended by a *difference metamodel* (MMD). This allows the evolution of the replica-versions to be captured in the difference models themselves, rather than as separate artifacts.

Operation Extraction

Having turned initially to state-based models, the workflow now moves into the operation-based realm. For each delta model, this phase assembles a set of change operations that accounts for that model's changes of state with respect to the base model. These *change sets* provide the raw material for the operation-based merge.

Conflict Detection

The extracted sets of change operations may now be combined, but their dependencies must be discovered before a full merge can be achieved. By assessing *operation dependencies*, this phase uncovers change precedence (e.g., one change is a precondition for another) and contradiction (e.g., one change undoes, or overwrites, another). The first imposes a partial ordering on the changes, while the second identifies conflicts between them that must be reconciled.

Conflict Resolution

Resolving a conflict entails picking one operation of a contradicting pair over the other, or rejecting both. Ideally, conflicts are *resolved automatically*; failing that, manually or, in the worst case, from the standpoint of correctness, arbitrarily. Making a conflict choice can alter change operation dependencies and, consequently, force a reordering of the changes. Once the extracted change sets are combined, their conflicts resolved, and elements ordered, a *merged operational trace* is outputted from the phase.

Model Patching

The changes of the merged operational trace are next *executed against the base model* to produce the merged outcome. Applying change operations to a model in the normal-form results in a transformed model, also in the normal-form. The merged result is thus state-based and will contain *no* difference metamodel elements, since there were none in the base model to begin with, and no operations in the merged trace create any MMD elements. Patching, then, is the means by which the workflow is returned to the state-based approach.

Model Denormalization

This phase effectively *reverses the work of the input phase*, transforming the merged model expressed in normal-form into the format(s) of the original input—one merged model being outputted for each type of input artifact. This means the originating tool(s) of the replica-versions will be able to pick up the newly merged version and manipulate it like any other artifact it may have created, even if the original models came from different (state- and/or operation-based) tools.

5.2 Workflow Phase Details

In this section, the phases of the hybrid merge workflow outlined above are detailed with the help of a basic set of algebraic operators for performing model management tasks, as proposed by Brunet et al. [2006], and extended by us where needed. The basic data types of the algebra are the *model*, the *relationship* between models, and the *transformation* of a model. The overall *merge* operator is defined as:

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

The operator assumes that the models to be merged are of the same type (e.g., the same kind of UML model) or of compatible subtypes. A model's subscript indicates its metamodel, with no subscript understood to be the normal-form. The relationship operand gives a mapping between elements of the two model operands, *not* how the models relate to each other as a whole. Knowledge of that sort becomes important if there are different ways of putting together an arbitrary pair of models, a possibility that falls outside the scope of this research.

The output of the merge operator and the end goal of the hybrid merge workflow are one and the same—a merged model. In what follows, each workflow phase introduces a new operator that achieves its function, and indirectly supports the overall goal of model merging.

5.2.1 Model Normalization: Decouple from Environment

Until now, a major drawback of the operation-based approach to merging has been overly tight coupling between the tool performing the merge and the change recording mechanism tracking the model changes. The intent of this phase is to break this coupling through a tool-independent means of representing the models. This allows models from a variety of modeling tools, whether state-based or operation-based, to be read in and merged—conceivably, even merging those from different tools.

Since the workflow does three-way merging, the expected input is three models: a common base, its left replica-version, and its right replica-version. The base model is not strictly necessary; if it is null, the merge becomes effectively a two-way merge. The path of execution through the workflow is the same regardless.

Operator *normalize* converts $model_{tool}$, a tool-specific model conforming to the tool's metamodel, into *model*, a tool independent model conforming to the metamodel of the normal-form.

$$\textbf{normalize} : model_{tool} \rightarrow model$$

The operator is overloaded to accept the models produced by different modeling tools. The unnormalized form of the input models may be state-based or operation-based, while the normalized form of the output is state-based.

An advantage to being state-based for the early phases of the workflow is that, because the complete state is at hand, two problems of the operation-based approach are avoided: *missing comparison* and *missing state* [Westfechtel, 2010]. The first problem can result in unnoticed, duplicate operations, and the second can lead to undetected conflicts.

Change operations obtained from state, however, are usually incomplete, because one change of state can encompass many change operations. But this is not a serious detriment, since many changes executed by a Modeler are of no importance for merging. It is *how* the models have changed and *not the exact path* by which they were changed that matters. The benefit is, changes extracted from state exhibit no redundancy. This results in fewer operations overall, and saves downstream processes a great deal of bookkeeping—two reasons state-based analyses and manipulations tend to be easier.

5.2.2 Model Comparison: Match Elements

The intent of model comparison is to find commonalities between models with which to establish correspondence between elements of the models [Treude et al., 2007]. The relationship thus divined is used as a basis for conflict detection, resolution, and model merging. The *match* operator as defined by Brunet et al. carries out this matching, or mapping, of elements.

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

Most tools rely on unique identifiers to accomplish matching, but this can only partially determine correspondence, because elements added separately to the left and right versions will not possess comparable IDs, though they may play the same role. Hence, the relationship between the models will usually be under-determined, with only elements having identical IDs being matched exactly. This conservative method of matching can leave a majority of elements without a match.

Using a *different matching strategy*, or *multiple strategies*, is a viable way of improving the quantity and quality of matches—something few tools provide for. Further, since no strategy is perfect, human intervention in the match process is often called for—something even fewer tools allow for. The approach used in this work is to employ several strategies and then either select the best one for each candidate pairing or use a weighted measure of similarity drawn from all the strategies. The matching that is done with these measures can be accomplished automatically or under human guidance.

5.2.3 Model Differencing: Capture Model Evolution

Given two models, it is common to determine how they are unlike one another by calculating the difference between their states. In the workflow, differencing of the models to be merged occurs after all models have been inputted and decoupled from their native environments. The *diff* operator as defined by Brunet et al. is:

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

The returned object is a transformation from the first model operand to the second, so the operator is not commutative. We break *diff* into two steps: a *delta* operator and an *extract* operator. This is done in order to better reflect progress through the workflow and to highlight that the deltas between the models are first-class artifacts in their own right. The *delta* operator is defined here, and *extract* in the next phase.

$\textbf{delta} : \textit{model} \times \textit{model} \rightarrow \textit{model}_\Delta$

The operator determines how the two models differ from one another, identifying which elements and properties have changed and in what way. This information is captured in a delta model conforming to a difference metamodel that is used to extend the metamodel of the normal-form. One delta is calculated between the base model and its left replica-version, and another between the base and its right replica-version. This is in preparation for transitioning back to operation-based activity, and is in contrast to the classic state-based approach, which performs a difference between the two replica-versions.

The delta models will contain instances of MMD classifiers detailing which elements have been added, which deleted, and which altered. Finding these differences from state-based model representations is relatively straightforward. Since most modeling tools assign an ID to their elements, and because the left and right versions begin life as replicas of the base model, a continuity of identity exists from common base model to its changed replicas. Thus, ID is sufficient to relate the elements in the replica-versions to their counterparts in the original, and a direct comparison can be used to determine their differences.

5.2.4 Operation Extraction: Transition to Operation-Based

With the relationship between the two replica-versions, and the deltas from their common base in hand, the hybrid workflow transitions from the state-based to the operation-based approach. Making this transition requires somehow gaining access to the change operations that were executed in modifying the base model replicas.

Unfortunately, any information that could be used to directly construct such an ordered list of change operations, or operational trace, has since been obliterated by the decoupling that takes place at the front of the workflow. While it is not possible to reassemble the *exact* series of operations that went into causing a specific change of model state, it is possible to determine *one* operation that subsumes the exact series, while effecting the same change in state. At some later time, a partial ordering of these subsuming operations can be worked out by examining the dependencies between them.

This situation is not the terrible thing it may at first seem to be, since even in operation-based merging there is seldom any need to know the exact chain of change operations executed against a model element before arriving at its final state (e.g., names given to a class before its final name are, except perhaps for matching purposes, unimportant to the merge). That the normalizing step results in fewer change operations can actually be *viewed as an advantage*, because it makes it computationally easier to reorder operations, and being able to vary execution order (e.g., to avoid an inconsistency) is one of the strengths of the operation-based approach.

To construct a list of change operations (in the form of a transformation), we define an *extract* operator:

$$\textbf{extract} : model_\Delta \rightarrow transformation$$

Given the difference between some initial model state and some final model state expressed as a delta model conforming to the difference metamodel, the operator produces a transformation that will take the model in the initial state to the final state. The delta model is of course obtained using the *delta* operator, such that:

$extract(delta(m_i, m_f)) = diff(m_i, m_f)$,

with m_i and m_f being the initial and final models, respectively.

The transformation thus produced is actually a composition of subtransformations, or change operations, as explained in Section 2.9. The difference metamodel makes the extraction of the operations a fairly straightforward process. Each `Added` and `Deleted` class found in the delta model results in one add and one delete operation, respectively. The `Altered` classes, however, require an in-depth comparison of the updated element with its original to see exactly how they differ, and then the creation of a change operation for each difference noted.

The order of the operations in the transformation produced by *extract* is determined solely by the order in which the elements of the difference model are visited, and has no significance as of yet. Once a transformation has been constructed for each side of the merge, the workflow proceeds in operation-based fashion. It is up to the next phase to impose the necessary ordering on the constructions.

5.2.5 Conflict Detection: Test Operation Commutativity

We define the *detect* operator to take two conflict-free transformations, in any order, along with a relationship (which may be empty). The relationship is the result of applying *match* to the models that are produced by transforming the base mode with the two transformation operands.

$$\textbf{detect} : transformation \times transformation \times relationship \rightarrow transformation$$

The operator results in another transformation composed of all the subtransformations of its operands. Conceptually, the operator applies the *before* predicate described in Section 2.9 to every pairing of *different* change operations (i.e., mirror pairs are ignored) in the two transformation operands. This exhaustively identifies all the conflicting pairs of operations. Pairs of conflicting *atomic operations* are then composed into single *compound operations*.

Predicate testing also imposes a partial ordering on the operations. Being partial, the operator has the freedom to move conflicts, respecting their dependencies, as close as possible to the terminating end of the resulting transformation. This has important ramifications for conflict resolution (Subsection 4.1.1). Of course, since a transformation is a functional composition, it must be totally ordered, so any operation not already placed by *before* must be given some arbitrary position.

For changes made to the same model, i.e., *same-side changes*, the input transformations are identical, and the relationship is a perfect one-to-one mapping. The resulting output will have order, but no conflicts, which is another advantage of the state-based approach: changes extracted by state from a consistent model can never be in conflict. This is because any particular change of state only ever results in one operation, and it takes two operations to make a conflict. For changes made to opposite models, i.e., *cross-side changes*, the transformations are different. Here, the relationship operand is used to determine if a particular pair of change operations in fact affect the same model element, and therefore could be in conflict.

5.2.6 Conflict Resolution: Eliminate Contradictions

The merged transformation from the previous phase may contain conflicting pairs of change operations, which this phase resolves. We use the *resolve* operator to represent this action:

$$\textbf{resolve} : transformation \rightarrow transformation$$

How the conflicts are to be dealt with is not stipulate by the operator—only that they will be resolved. After any automatic resolution, the change operations of the transformation are arranged one last time in an attempt to push any remaining conflicts further back. This will allow the most complete merge attainable, with as much context as possible to be presented to the user for manual resolution of the remaining conflicts. Once this is accomplished, the change operations are composed into the final merged transformation.

5.2.7 Model Patching: Execute Merged Operations

To return to the state-based world, the change operations of the merged transformation are executed against the normal-form base model to produce the merged model, also in the normal-form. Brunet et al. define a *patch* operator for this purpose.

$$\textbf{patch} : model \times transformation \rightarrow model$$

If the transformation still contains conflicts, the leading subtransformations up to, but not including the first conflict are run against the model operand. Earlier phases moving conflicts as near to the end of the transformation as they can, ensures that *patch* will be able to produce the most complete merge possible before encountering a conflict.

5.2.8 Model Denormalization: Restore to Environment

The last phase of the workflow transforms the final merged model back into its original form. This is the function of the *denormalize* operator, which we define as:

$$\textbf{denormalize} : model \rightarrow model_{tool}$$

The operator takes a model expressed in the tool-neutral normal-form and transforms it into one conforming to the metamodel of the original tool's inputs to the workflow. If models of more than one tool were inputted, *denormalize* is executed once for each tool type, producing multiple but logically equivalent models (within limits of the involved metamodels).

That completes the theoretical description of the proposed hybrid merge workflow and its phases. The next two chapters give overviews, and some of the details for the architecture and design of a prototype that implements the workflow.

5.3 Workflow Summary

Though we have incorporated several new implementation features into them, most of the phases described in the preceding two sections are not unique to the hybrid workflow. This section will serve to clarify these aspects, and to provide pointers to the specific phase details.

Model Normalization and Denormalization

These two phases are unique to the workflow. They decouple the models to be merged from their originating tools, allowing the workflow to deal with (initially) state-based models, expressed in the same format, and conforming to the same metamodel. Their architecture is described in Subsection 6.3.1, and their design in Subsection 7.2.1.

Model Comparison

All merging tools have to relate the model elements on one side of the merge to those on the other. The control interface of this phase enables what no existing tool allows—influence of this mapping by the user. Also novel, is the use of multiple matching strategies in the implementation. Subsection 6.3.2 and Section 7.3 detail the comparison phase's architecture and design, respectively.

Model Differencing

Finding the difference between models is an important step in state-based merging, but not operation-based. Representing these differences as models appears to be unique to the hybrid workflow: we know of no other merging tool that can make a similar claim. Subsection 6.3.1 covers the architecture, and Subsection 7.2.2 the design, and includes a description of the difference metamodel.

Operation Extraction

Again, this phase is a standard step in state-based merging, where the extracted entities form two *sets* of updates. The workflow, in anticipation of transitioning to operation-based behavior, considers them to be subtransformations, out of which it composes two *transformations*. Architecture of the extraction phase is covered by Subsection 6.3.3 and its design by Subsection 7.4.2.

Conflict Detection

Conflict detection is a requirement for merging. Several techniques were presented in Section 2.8, but the phase does not prescribe any particular one. While order is implicit in the operation-based approach, classic state-based merging does not make use of it. The workflow, uniquely uses detection to double as a way of ordering the operations of its input into one merged transformation. The order established by the phase is generally only partial, and is exploited to move conflicts towards the end of the output transformation, which has benefits for useability. Subsection 6.3.3 discusses the architecture, and Subsection 7.4.2 the design of this phase.

Conflict Resolution

All merge tools must resolve conflicts, and most provide only a manual means for doing so. This phase adds nothing new in this regard. Any uniqueness is confined to the implementation, which uses decision tables to enable semantic conflict detection and resolution. The phase's architecture is found in Subsection 6.3.3, and its design in Subsection 7.4.2. Details of the decision table mechanism and its rules are part of Subsection 7.4.3.

Model Patching

Patching is a common model management task. It is executed in one form or another by both merging approaches. Subsection 6.3.3 of the architecture touches on patching, but it is not specifically covered in the design.

Chapter 6

Mirador Architecture

Mirador is a model merging tool that has been built as a proof of concept to validate the proposed hybrid merge workflow of Chapter 5. How Mirador's architecture supports the workflow is the subject of this chapter. The fact that it is a vehicle for describing an existing system, rather than a design document, allows us some latitude in selecting a perspective for explaining the architecture. Accordingly, the perspective switches freely between being structural in nature and taking a runtime point of view, as needed. Since Mirador is a prototype, the implementation is naturally somewhat limited in scope. These limitations are pointed out in the course of the presentation.

6.1 System Purpose and Environment

This section gives an overview of Mirador's intended usage through its primary use case. It also sets the operational expectations the system has of, and the impact it has on, its environment.

6.1.1 System Context

Mirador is a stand-alone application for merging software models. Because it *is* stand-alone, the tool is independent and self-contained; as such, its interactions are limited to one user at a time. The system boundaries—"what is in, and what is out," in the words of Clements et al. [2003]—are clarified by the system context diagram of Figure 41. The environment depicted consists of two human actors, as well as the

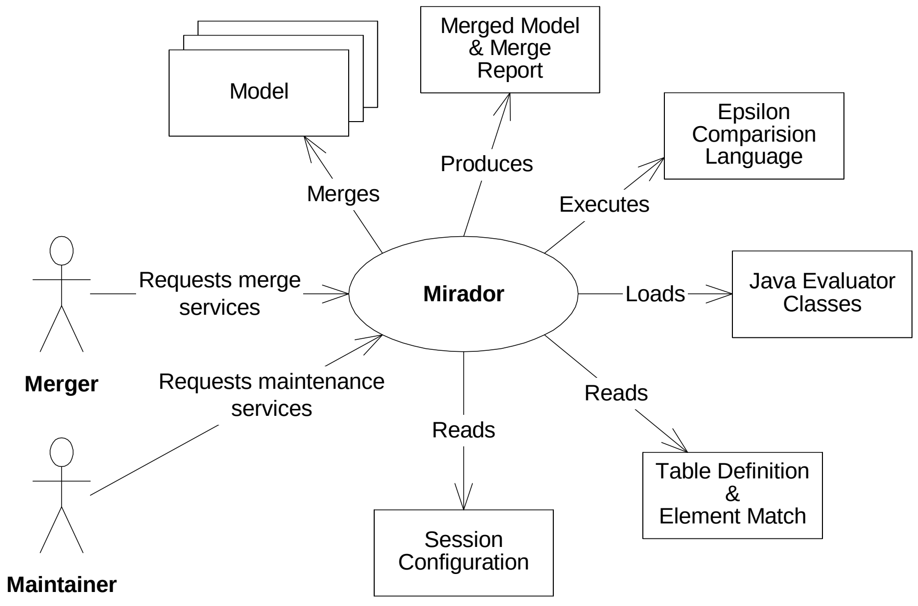


Figure 41: Top-level system context diagram delineating Mirador's boundaries

many file types consumed and produced by the application at different stages of its execution.

6.1.2 System Interfaces

Mirador interfaces with no external systems other than the Java virtual machine on which it runs. In addition to supplying the runtime platform, Java provides Mirador with interfaces for accessing services of the file system and a character-based terminal, and for creating and interacting with an event-driven graphical front-end that supports the tool's interactive mode of operation.

6.1.3 User Interfaces

In interactive mode, Mirador works with a single user through a graphical user interface consisting of three panels: Model Input, Element Match, and Model Merge. The panels roughly correspond to important subsystems of the tool and may be seen in Figures 51, 56 and 58 of Chapter 7.

The first panel inputs the models to be merged. Options for the new merge session may also be set with this panel. Besides the models, the panel also may read in a configuration file, files containing model element matches, match strategy classes, and decision table definitions. The second panel is concerned with matching the elements of one replica-version model with those of the other. The panel provides the user-access point into the Model Comparison phase of the workflow. The last panel uses these pairings to merge the changes and resolve any resulting conflicts. It provides the Conflict Resolution workflow assess point.

In non-interactive, or batch, mode, all inputs are specified on the command line, or through a configuration file. This mode is useful for running group or unattended merges from a script, or as a means of getting early feedback on a merge with minimal effort. Unlike the interactive mode, this mode has no provisions for manually affecting element matching or conflict resolution.

The output from the interactive and batch modes is a merged model. The model that is output from batch mode however, is more likely to be only partial, due to conflicts that could not be resolved automatically. In addition, both modes also generate a textual merge report.

6.1.4 Software Interfaces

Mirador interfaces to other software through character-based files. In the case of models, this is usually some variant of XMI exported by an application (e.g., .ecore model files from the Eclipse Modeling Framework), or files in an application-specific format (e.g., .ctr model files created with Fujaba).

The formats of the textual configuration (.cfg), element matching (.elm), and decision table definition (.ddf) files are all specific to Mirador. To help with element matching, Mirador can also load user-defined Java modules (.class) and scripts written in the Epsilon Comparison Language (ECL) for use as similarity evaluators.

6.2 System Functionality

While this is not a requirements document, for purposes of traceability, we have elected to express application functionality with use cases that interact with domain model entities. A useful dimension of refinement for use cases is goal detail, which may be at one of three levels, from highest to lowest: *summary-level goal*, *user-level*

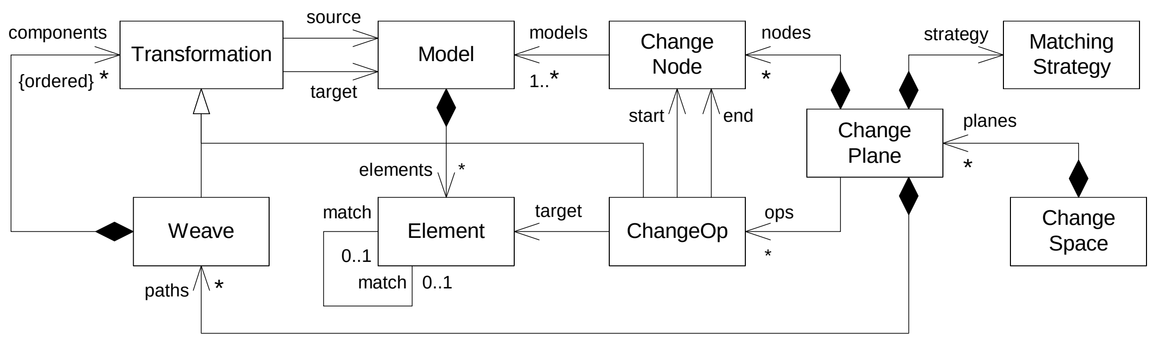


Figure 42: Mirador domain model

goal, and *subfunction-level goal* [Cockburn, 2001]. For this work, the only use case details that will concern us will be those at the user-goal level.

6.2.1 Domain Model

Actors interact with use cases, and use cases, in turn, access and manipulate objects from the problem domain, i.e., the universe of entities specific to the application. The classes for these objects and the relations that may exist between them are captured by the domain model that appears in Figure 42.

The domain model shows how the central notion of a **Model** is composed of many model **Elements** that are placed on a **ChangePlane** within a **ChangeSpace**. Multiple **ChangePlanes**, each having its own **MatchingStrategy**, occupy the **ChangeSpace**, and each **ChangeNode** on a plane hosts one or more **Models**.

A *source* model is changed into a *target* model by transforming the source with a **Transformation**. A transformation may be an individual **ChangeOp**, which *starts* at one node and *ends* at an adjacent one, or a composition of change operations, known as a **Weave**, representing a path on the change plane.

A **ChangeOp** targets one model **Element** that it changes in some way. A bidirectional *match* association between elements on opposite sides of the merge, is used to indicate those that have been identified as being the same, and should always occur in reciprocal pairs, i.e., if one element is matched with a second, the second should be matched with the first.

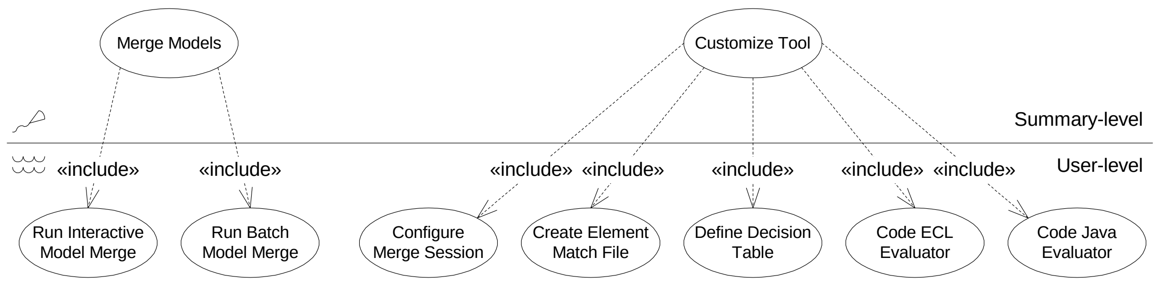


Figure 43: Mirador summary use cases

6.2.2 Mirador Use Cases and the Primary Actors

The summary-goal use cases for Mirador are shown in Figure 43, along with the user-goal use cases they encompass (through *include* associations). Summary use cases provide the context in which the lower level use cases operate and serve as an index into them. Most of Mirador's notable behaviors and features are illustrated with the "Run Interactive Model Merge" user-goal use case. The scenario, which touches upon the most salient design points of the tool, has been written out below in full use case prose. The text of the "Run Batch Model Merge" use case is given in Appendix B.

The system's user-goal use cases and primary actors are pictured in Figure 44. Primary actors are those actors with a goal that requires the system's assistance to achieve. In general, primary actors are the initiators of use cases. Each user-goal use case consists of a set of paths (*scenarios*) that move its primary actor from a trigger event that starts the use case to the goal (*success scenarios*) and other scenarios that move the actor toward the goal but fall short of reaching it (*failure scenarios*).

Actor: Merger

The Merger is responsible for bringing two model versions into alignment and, in the process, producing a new merged version of the model. Besides interacting directly with the tool, the Merger may also need to analyze merge reports to learn how best to execute a particular merge.

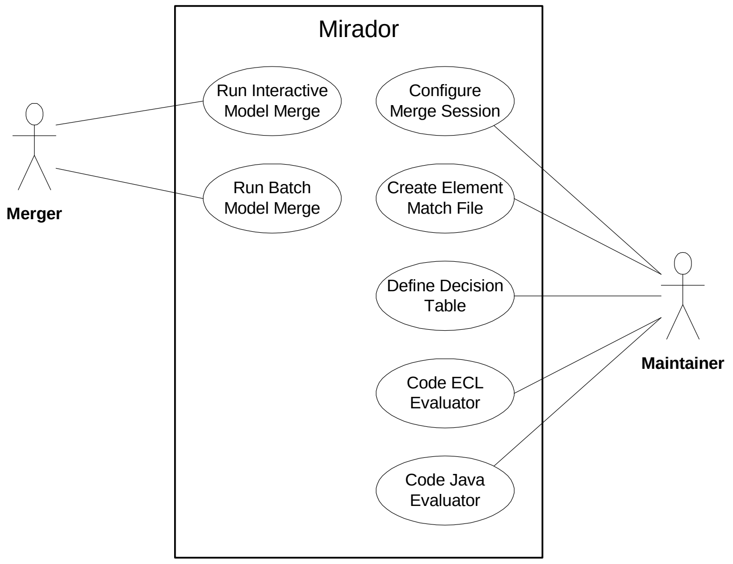


Figure 44: Mirador user-goal use cases

Use Case: Run Interactive Model Merge

Goal: To merge models with input from the user.

Primary Actor: Merger

Precondition: none

Main Success Scenario

1. Merger invokes Mirador, specifying desired arguments and any options.
2. System displays Model Input Panel, populated with argument and option values.
3. Merger enters/alters session parameters, and advances to Element Match Panel.
4. System sets up session based on passed arguments and options, and loads specified models.
5. System displays models to be merged and their matched elements.
6. Merger selects matching strategy, and adjusts weights and threshold.
7. Merger inspects, matches or unmatches elements, and advances to Model Merge Panel.
8. System displays base model and an ordered list of change operations and conflicts.
9. Merger selects changes to apply, resolves conflicts and applies the change set.
10. System patches base model with change set and updates display with merged model.
11. Merger advances to merge finish.
12. System outputs the (partially) merged model in the format(s) of the original[1].

    Use case ends successfully.

[1]Not yet supported. Currently, all models are output as Ecore XMI files regardless of input type.

Extensions

1a. **A configuration file is specified on the command line.**

  1a1. System parses configuration file arguments and options.

  Use case continues.

3a. **A specified file is not found, or cannot be parsed.**

  3a1. An error is displayed and logged.

  Use case ends unsuccessfully.

3b. **The model format is not supported, or the model is not well formed.**

  3b1. An error is displayed and logged.

  Use case ends unsuccessfully.

7a. **Only two models were loaded.**

  7a1. A 3-way merge using a null base model is performed.

  Use case continues.

10a. **An error is encountered while applying a change.**

  10a1. Change application halts, and the change is flagged as being in error.

  Use case continues.

Postcondition: Merged model and merge report have been stored.

6.3 Architecture Highlights

The use case of the preceding section disclosed the major aspects of Mirador's functionality. In presenting its architecture, the focus will be on communicating this functionality in intellectually manageable pieces, which can then be refined to convey more details. With this in mind, the description makes use of the *module viewtype*, specifically, the *module decomposition style*, to provide a top-down presentation of the system's responsibilities [Clements et al., 2003]. Details of the tool's implementation and runtime behavior are presented in Chapter 7.

The activity diagram of Figure 45 models the logic of the "Run Interactive Model Merge" and "Run Batch Model Merge" (Appendix B) use cases. The tasks performed by the system in the course of merging two models have been divided into three partitions, corresponding roughly to the application's chief subsystems and exactly to the panels of its graphical user interface. Each partition has an iconic key in the upper right that highlights the phases of the hybrid merge workflow that its activities

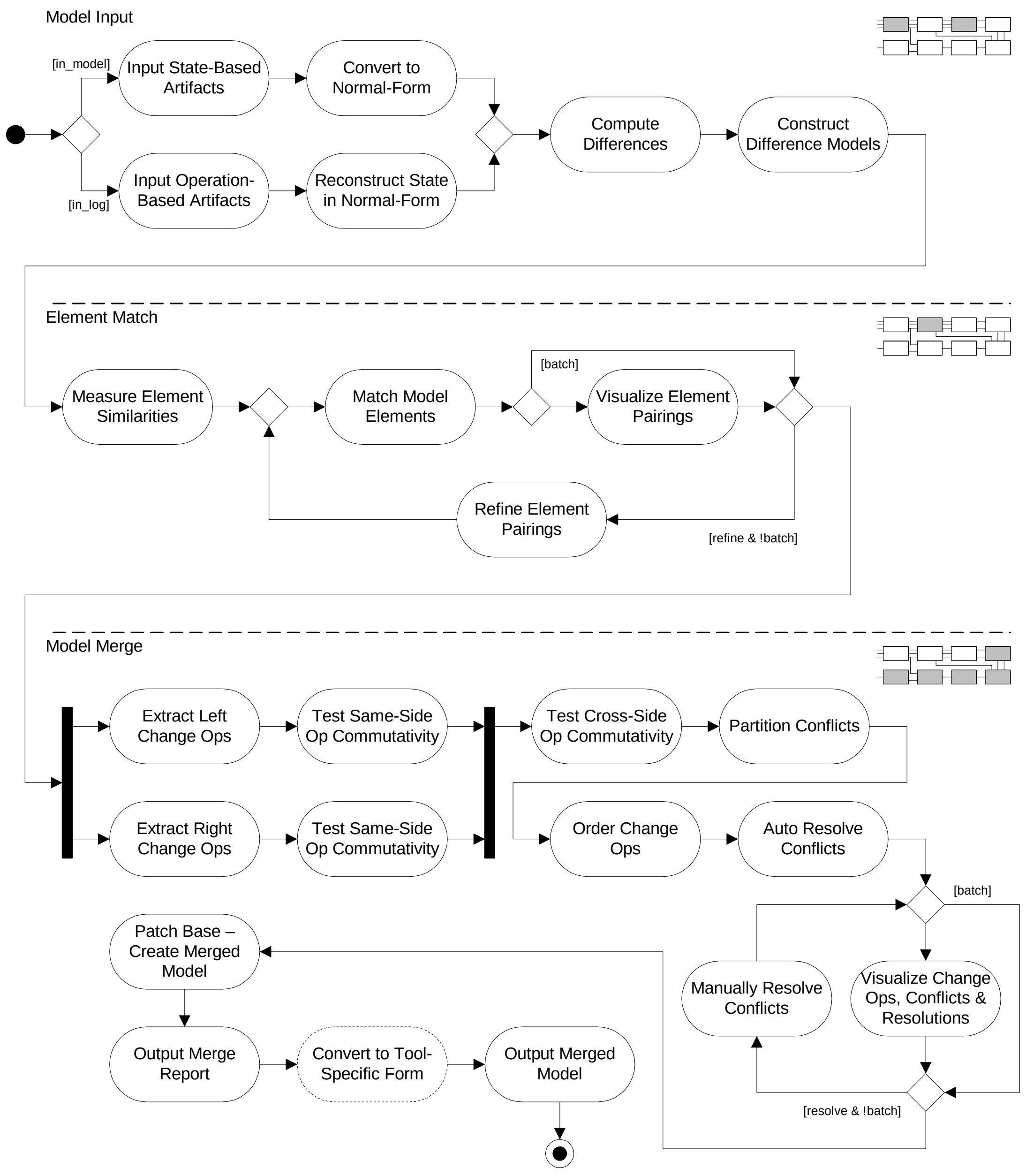

Model Input
[in_model]
[in_log]
Input State-Based Artifacts
Convert to Normal-Form
Input Operation-Based Artifacts
Reconstruct State in Normal-Form
Compute Differences
Construct Difference Models
Element Match
Measure Element Similarities
Match Model Elements
[batch]
Visualize Element Pairings
Refine Element Pairings
[refine & !batch]
Model Merge
Extract Left Change Ops
Test Same-Side Op Commutativity
Extract Right Change Ops
Test Same-Side Op Commutativity
Test Cross-Side Op Commutativity
Partition Conflicts
Order Change Ops
Auto Resolve Conflicts
[batch]
Manually Resolve Conflicts
Visualize Change Ops, Conflicts & Resolutions
[resolve & !batch]
Patch Base – Create Merged Model
Output Merge Report
Convert to Tool-Specific Form
Output Merged Model

Figure 45: Activity diagram for interactive and batch model merging in Mirador

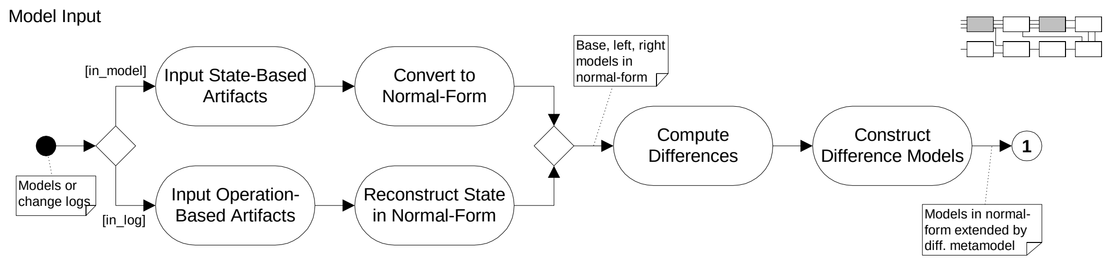

Model Input
[in_model]
Input State-Based Artifacts
Convert to Normal-Form
Base, left, right models in normal-form
Compute Differences
Construct Difference Models
1
Models or change logs
[in_log]
Input Operation-Based Artifacts
Reconstruct State in Normal-Form
Models in normal-form extended by diff. metamodel

Figure 46: Model input partition of Mirador activity diagram

relate to. The partitions are reproduced separately, in a larger size and with added annotations, in the immediate subsections.

Input to Mirador for a three-way merge is the base model and its left and right replica-versions. If the base model is null, the merge is still three-way with the process proceeding in exactly the same manner. There will be no add-delete problems if the base model is truly null, as opposed to being unknown, or if the versions really share no common ancestor, as when two disparate subsystems are being merged.

6.3.1 Model Input

The topmost partition of the activity diagram in (Figure 46) decouples the input models from their modeling tool in the case of state-based models, or from their change recorder in the case of operation-based models (i.e., change logs). The decoupling is effected by an **Input** subsystem that has an interface through which a **Decoupler** class for any supported type of model or change log may be reached. The proper class is chosen based on characteristics of the model being inputted.

As Figure 47 indicates, the decouplers reference a metamodel in transforming an input model or change log into an equivalent state-based model in the normal-form. Using the services of tool-specific classes of the **Repository** package, the elements of a state-based model are mapped to the correct representations, while the changes of operation-based logs are essentially executed to create the equivalent elements. In this way, the inputted models/logs are normalized to a common metamodel. After decoupling Mirador operates exclusively with state-based artifacts until the middle phases of the hybrid workflow.

Class **DifferenceCalculator** is used by the decouplers to compute the difference between the base model and its left version, and between the base model and its right

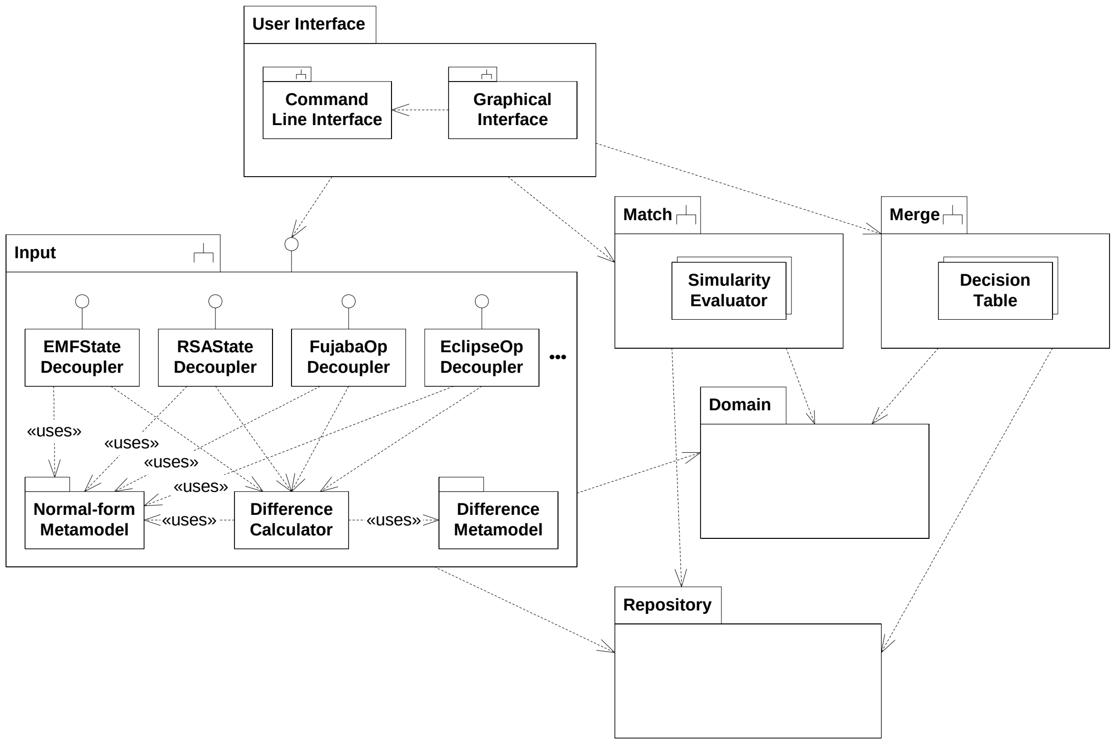


Figure 47: Module decomposition of the Mirador model merger

version. Two *difference models*, conforming to the MMD extended normal-form, are constructed in the process. These are stored in a nonpersistent class of the repository.

The **User Interface** package supplies the models to be merged for normalization to the **Input** subsystem through a facade interface. The input may come indirectly from the **Graphical Interface** subsystem, or directly from the **Command Line Interface** subsystem, on which the former relies. The two avenues are the same as far as the **Input** subsystem is concerned. The subsystem also depends on tool-specific variants of the **ChangeOp**, **Element** and **Model** classes that are part of the **Domain** package.

6.3.2 Element Match

The middle lane of the activity diagram (Figure 48) addresses the matching of elements between the two difference models. It corresponds to the **Match** subsystem seen in Figure 47. If the system is being run in batch mode, there is no interaction between the subsystem and the **User Interface**. However, running in interactive

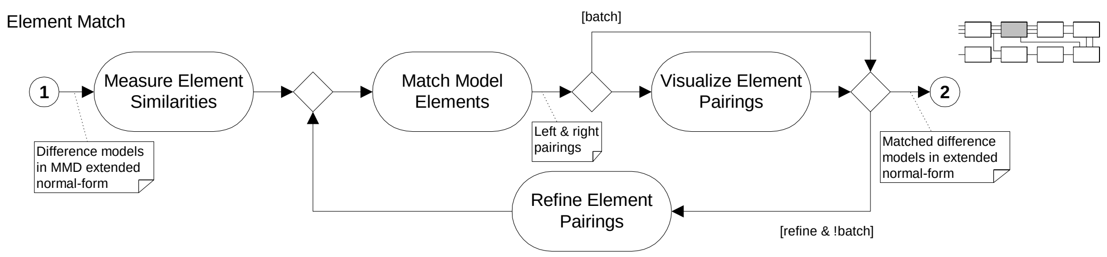

Element Match
[batch]
1
Measure Element Similarities
Match Model Elements
Visualize Element Pairings
2
Difference models in MMD extended normal-form
Left & right pairings
Matched difference models in extended normal-form
Refine Element Pairings
[refine & !batch]

Figure 48: Model match partition of Mirador activity diagram

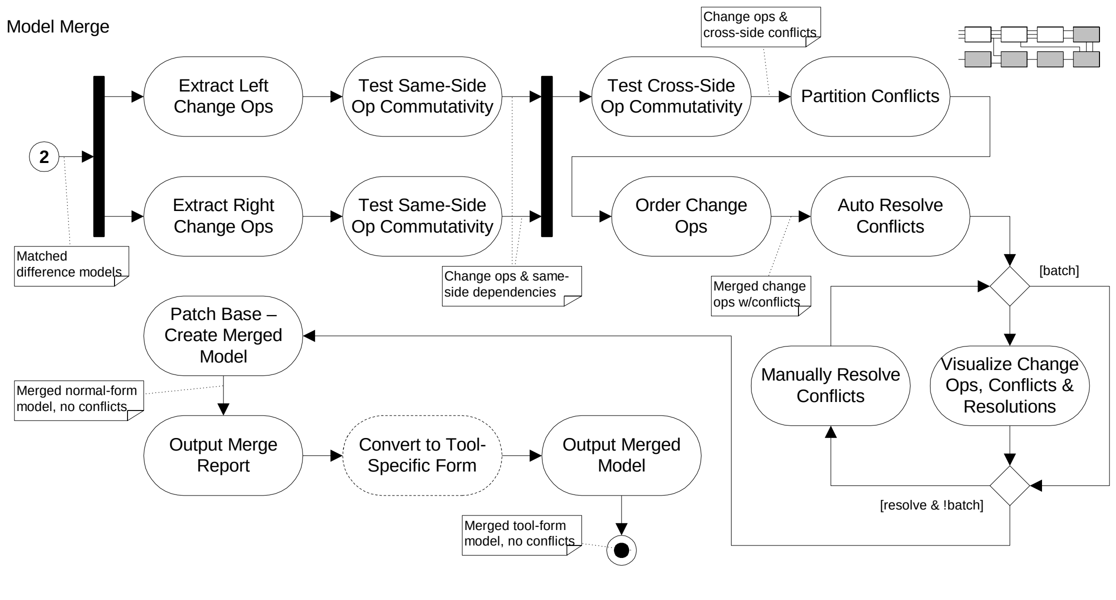
mode results in a great deal of communication between the two.

The first thing the **Match** subsystem does is to measure the similarity between all pairs of like elements across the difference models. The models themselves are obtained from the **Repository** package. Each loaded **MatchStrategy** of the domain model is associated with a **SimilarityEvaluator** class, which is responsible for actually conducting the similarity measurement for its strategy. A measure of overall similarity is also made. This is the weighted sum of the individual measures for a given model element pair. The weighting for each strategy may be set from either the graphical or command line interfaces.

Model elements are paired automatically based on the selected strategy and a minimum threshold value. The graphical interface allows for the automatic matching to be refined by the user as much as desired. Any manually made or unmade matches will not be undone by automatic matching, but must be explicitly released by the user. The final pairings that result from the cycles of automatic and manual matching are preserved with the *match* link of the domain model's **Element** class.

6.3.3 Model Merge

It is the last partition (Figure 49) that manages the transition from state-based to operation-based activity and back again. The front transition occurs with the extraction of one set of change operations from each of the two difference models found in the **Repository**. The extracted operations, along with the matching strategies selected for the session, are used to populate the **ChangeSpace** of the domain model. Testing for same-side commutativity with the decision table-driven *before* predicate then imparts a partial order, based on their dependencies, to the changes. Recall that change operations extracted from a well-formed, state-based model never conflict.



Model Merge
2
Matched difference models
Extract Left Change Ops
Extract Right Change Ops
Test Same-Side Op Commutativity
Test Same-Side Op Commutativity
Change ops & same-side dependencies
Test Cross-Side Op Commutativity
Change ops & cross-side conflicts
Partition Conflicts
Order Change Ops
Merged change ops w/conflicts
Auto Resolve Conflicts
[batch]
Manually Resolve Conflicts
Visualize Change Ops, Conflicts & Resolutions
[resolve & !batch]
Patch Base – Create Merged Model
Merged normal-form model, no conflicts
Output Merge Report
Convert to Tool-Specific Form
Output Merged Model
Merged tool-form model, no conflicts

Figure 49: Model merge partition of Mirador activity diagram

The outcome of the extraction is two partially ordered operational traces of change operations—one trace for each side. Application of the *before* predicate to the changes of *both* of these traces produces yet another trace. This one is made up of cross-side operations, partially ordered by their interdependencies, which may be contradictory. The result is actually a *merged* transformation, because it is an ordered composition of subtransformations from both sides. Change partitioning and ordering, as covered in Subsection 4.3.2, along with table-driven resolution, are then used to automatically reduce the number of conflicts and move any unresolved ones to the end of the trace. Eliminating all conflicts usually requires manual resolution, which in interactive mode is accomplished through the **Graphical Interface** subsystem.

Patching the base model with the operations of the merged transformation produces the final merged model. The product is a state-based model in the normal-form and transitions the workflow back to the use of state-based artifacts. No MMD elements will be present in the merged model, because none exist in the base model, and none are produced by execution of change operations—only by model differencing.

After patching, a textual report of the merge session is written out. Then the **Decouplers** of the **Abstraction** subsystem are used in reverse to transform the normal-form merged model into its equivalent in the original tool-specific form(s).

This activity is drawn in the diagrams with a dashed line to indicate that Mirador does not yet fully support this feature. Currently, all merged models are outputted as Ecore models in an XMI format. While not ideal, this has nevertheless proven sufficient for validating the idea of a hybrid merge workflow (Chapter 8).

With Mirador's primary behavior expressed and key portions of the system architecture explained, there is now sufficient context for giving some details of the system's design—the topic of the next chapter.

Chapter 7

Mirador Features and Design Highlights

This chapter functions as an abridged technical user's guide by using scenarios of the "Run Interactive Model Merge" (Subsection 6.2.2) and "Run Batch Model Merge" (Appendix B) use cases to illustrate the runtime behavior of Mirador and introduce several of its features. Over the course of the presentation, select design aspects are also highlighted. Because they are loosely associated with the tool's primary subsystems, and reflect their internal workings, the main panels of the GUI will serve to frame the discussion. Not having a GUI, batch mode is addressed separately.

The tool runs stand-alone on the Java 6, Standard Edition platform. When running in interactive mode, it is a graphical event-driven application that houses its three main panels in a tabbed dialog. In batch mode, it runs as a command line-driven application. Mirador may also be run interactively as a plug-in of the Fujaba MDD tool suite [Fujaba, 2009].

7.1 A Running Merge Example

To better illustrate tool operation, a more involved and complete example than the one given in Chapter 4 is defined here. It is a simplified version of the "HTML Document" example used in [Cicchetti et al., 2007]. The class diagrams for the base, left and right models to be input for merging are arranged in Figure 50 in the usual configuration. The base model was created in Fujaba, after which the model file was

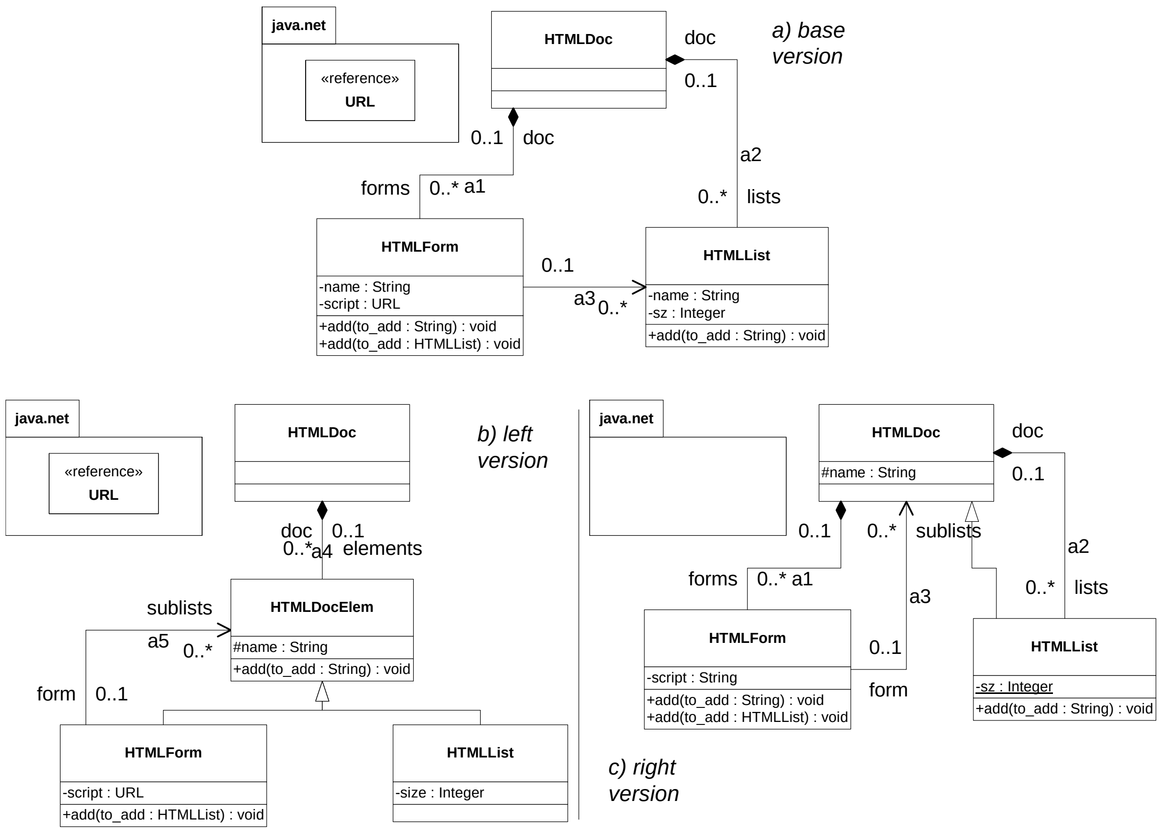


Figure 50: Three-way merge of "HTML Document" running example

twice replicated, and the replicas, again using Fujaba, then modified. A Fujaba model is, in reality, a change log maintained by the CoObRA (Concurrent Object Replication frAmework) versioning software [Schneider et al., 2004] that tracks model changes as a sequence of file records, grouped into transactions. So a Fujaba model is, in fact, an operation-based artifact.

A comparison of the models in the example shows that the left Modeler has, in no particular order:

- inserted class **HTMLDocElem** between the three original classes;
- made this new class the superclass for **HTMLForm** and **HTMLList**;
- pulled the `name` attribute and one of the `add` methods up into the new class;
- deleted all three associations and added two new ones; and
- renamed attribute `sz` to `size`.

Meanwhile, the right Modeler has, also in arbitrary order:

- made class **HTMLList** a subclass of **HTMLDoc**;
- pulled up attribute `name` into the superclass;
- deleted `name` from *both* of the other classes—likely a mistake;
- changed association `a3` to reference **HTMLDoc**;
- changed the type of `script` and removed the **URL** data type; and
- altered the lower bound of `sz` to 1.

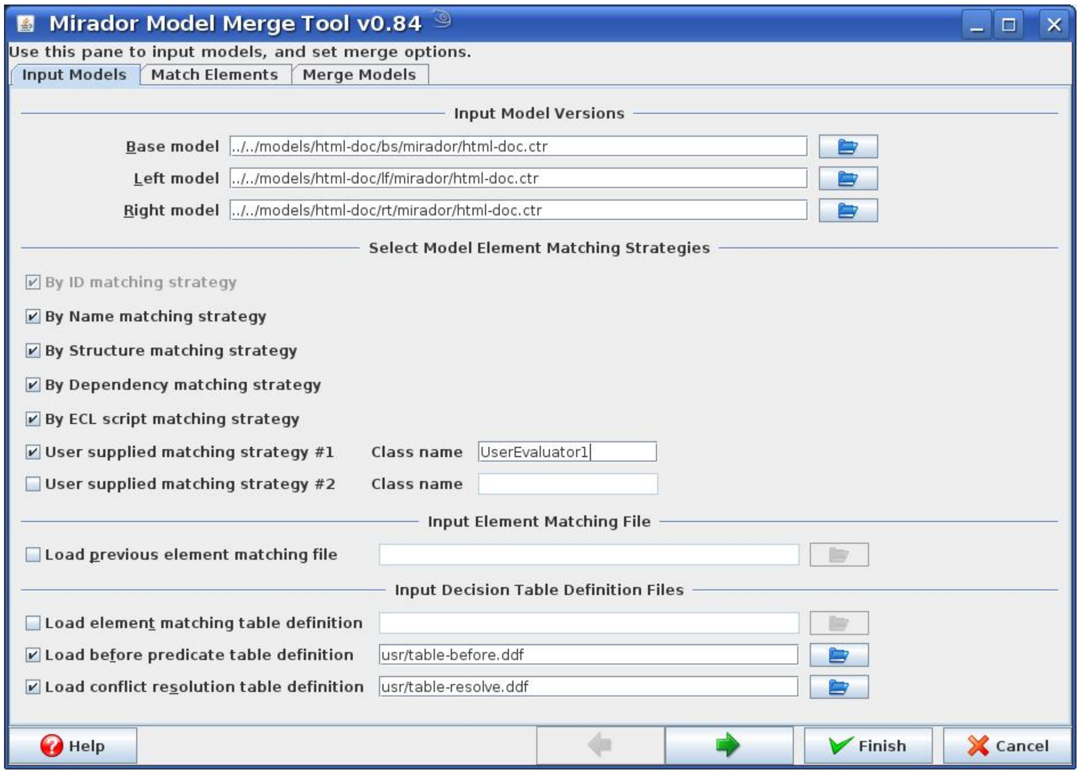
What Mirador does with these three models when asked to merge the left and right replica-versions is taken up in the remaining sections of the chapter.

7.2 Model Input Panel

When following the "Run Interactive Model Merge" scenario, Mirador opens up its tabbed interface to show the panel of Figure 51. The interface suggests a linear traversal order, but it is possible to move freely among the panels as long as their preconditions are met (e.g., models must be loaded before moving off this panel). Command line arguments and options, or a configuration file may be used to pre-populate fields and set controls on the panels. The syntax is detailed in Appendix C.

The Merger enters or alters information in four sections of the panel. The topmost section is where the file names and paths of the models to be merged are specified. The names filling the fields in the screenshot are all operation-based Fujaba models. Thanks to its decoupling, Mirador can actually load models of *different* MDE tools into the same merge session.

The next panel section allows for selecting which matching strategies will be used for establishing model element correspondence. This lets the Merger limit the computational costs of matching to only those strategies that they wish to pay for. The Merger may additionally elect to supply up to two of their own matching strategies in the form of Java class files.


Mirador Model Merge Tool v0.84

Use this pane to input models, and set merge options.

Input Model Versions

Base model ../../models/html-doc/bs/mirador/html-doc.ctr

Left model ../../models/html-doc/lf/mirador/html-doc.ctr

Right model ../../models/html-doc/rt/mirador/html-doc.ctr

Select Model Element Matching Strategies

☑ By ID matching strategy

☑ By Name matching strategy

☑ By Structure matching strategy

☑ By Dependency matching strategy

☑ By ECL script matching strategy

☑ User supplied matching strategy #1 Class name UserEvaluator1

☐ User supplied matching strategy #2 Class name

Input Element Matching File

☐ Load previous element matching file

Input Decision Table Definition Files

☐ Load element matching table definition

☑ Load before predicate table definition usr/table-before.ddf

☑ Load conflict resolution table definition usr/table-resolve.ddf

Figure 51: Mirador Model Input Panel

The third section of the panel lets the Merger name a file containing match information saved from a previous merge session. The intent of this file is to impart information about model matching circumstances that the Merger is aware of, but which the tool is not. This will help Mirador to make match decisions that are more likely to be correct, thereby streamlining the match process for the Merger.

The fields at the bottom of the screen let the Merger specify three decision table definition files[1] None are required, since Mirador has default versions, but they do open up much of the tool's operation to customization should that become desirable. This capability gives organizations that deal primarily with models of a specialized domain the opportunity to essentially create a custom model merging tool for their domain by defining the rules that apply in that environment.

Pressing the **Forward** button will, assuming no errors are encountered, advance the Merger to the next panel. Behind the scenes, the event also triggers a number of actions: normalized equivalents of the named models are built; configuration files are

[1]The element matching table is not currently supported.

read in and parsed; decision tables are constructed; and similarity evaluators for the selected and user-supplied matching strategies instantiated.

7.2.1 Decoupling Models from their Originating Tools

Before starting work with the input models, Mirador converts them to conform to Ecore [EMF Ecore Package, 2011], the state-based metamodel of the Eclipse Modeling Framework. This corresponds to the Model Normalization phase of the hybrid workflow (Subsection 5.2.1). Ecore was chosen as the defining metamodel for Mirador's *normal-form* due to its widespread usage and relative simplicity. All models internal to Mirador are instances of the normal-form, except for the difference models, which instead conform to an *extended normal-form*. This form is extended in the sense that it allows changes to a normal-form model to be represented within the model itself, thus becoming first-class artifacts in their own right. The extended normal-form is based on a difference metamodel that is described later. In the code, models of both forms are created as objects of the type **MiradorModel**.

Model decoupling is accomplished by the **Input** subsystem (Figure 47). The subsystem requires a dedicated module to do the conversion to normal-form for every model type of interest. So far, two interfaces have been coded: one for EMF, a state-based model type, and one for Fujaba, an operation-based model type. The key classes for both interfaces are shown in the class diagram of Figure 52. More details about the decouplers may be found in [Barrett et al., 2010b].

Instances of the **ModelRepository** hierarchy classes on the left side of the diagram act as converters for the input models of the merge, and caches for its working models. The abstract root has data members to hold six models: the normal-form versions of the base, left, and right input models (names have been shortened to save space); the left-minus-base and the right-minus-base difference models; and the yet to be merged final model. The **changes** data member holds the change operations that will be eventually extracted from the difference models.

The specialized classes of the hierarchy do the actual conversion of their model types to the normal-form. For the **EMFModelRepository**, since its models are already in normal-form by definition, this simply entails loading the model with its **loadEmfModel** method, which essentially does a one-to-one mapping from elements

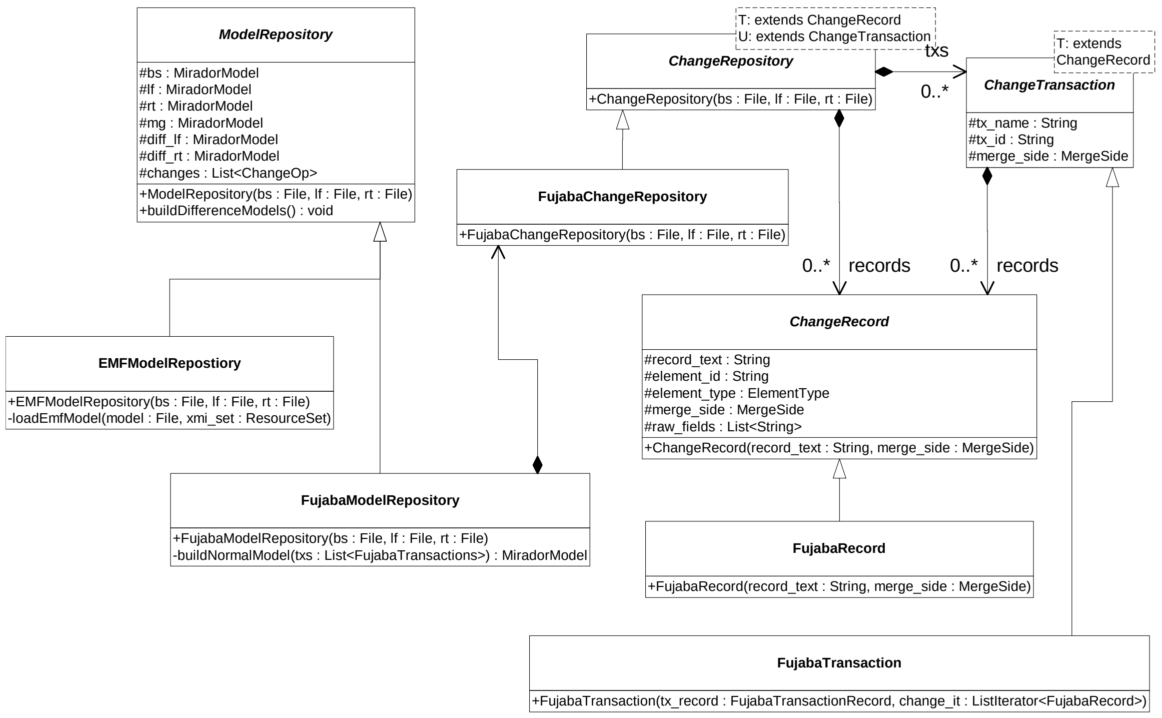

ModelRepository
#bs : MiradorModel
#lf : MiradorModel
#rt : MiradorModel
#mg : MiradorModel
#diff_lf : MiradorModel
#diff_rt : MiradorModel
#changes : List<ChangeOp>
+ModelRepository(bs : File, lf : File, rt : File)
+buildDifferenceModels() : void
EMFModelRepostiory
+EMFModelRepository(bs : File, lf : File, rt : File)
-loadEmfModel(model : File, xmi_set : ResourceSet)
FujabaModelRepository
+FujabaModelRepository(bs : File, lf : File, rt : File)
-buildNormalModel(txs : List<FujabaTransactions>) : MiradorModel
T: extends ChangeRecord
U: extends ChangeTransaction
ChangeRepository
+ChangeRepository(bs : File, lf : File, rt : File)
txs
0..*
FujabaChangeRepository
+FujabaChangeRepository(bs : File, lf : File, rt : File)
T: extends ChangeRecord
ChangeTransaction
#tx_name : String
#tx_id : String
#merge_side : MergeSide
0..* records
0..* records
ChangeRecord
#record_text : String
#element_id : String
#element_type : ElementType
#merge_side : MergeSide
#raw_fields : List<String>
+ChangeRecord(record_text : String, merge_side : MergeSide)
FujabaRecord
+FujabaRecord(record_text : String, merge_side : MergeSide)
FujabaTransaction
+FujabaTransaction(tx_record : FujabaTransactionRecord, change_it : ListIterator<FujabaRecord>)

Figure 52: EMF and Fujaba interfaces of the decoupling Input subsystem

of the model to classifiers of the metamodel. Other state-based interfaces would have more complicated mappings.

The **FujabaModelRepository** has a much harder time of it. Its conversion method, **buildNormalModel**, must first read and parse the lines of the Fujaba model into Fujaba **ChangeRecords** and group them into **ChangeTransactions** (Fujaba log record syntax is described in Appendix F). Then it must determine what effect the action of each change record has on its targeted element, and, finally, map these modified targets to the proper classifiers of the Ecore metamodel. This work is supported by the Fujaba specific classes appearing to the right of the repositories in Figure 52. A **FujabaChangeRepository** object functions as a scratchpad and an intermediary between change records and the repository.

Many of the transactions in a Fujaba model have straightforward mappings. Take the two CoObRA records shown here:

```
1  t;qHgwp#;inplace editing;1292857493253;-;;
2  c3;;i:DJJ3S#961;name;v::size;v::sz;-;i:qHgwp#;
```

The literal ‘t’ in the first, semicolon-delimited field of line #1 marks it as a transaction record. The second field contains its unique ID. The records that follow, up to the next transaction record or file end, belong to the same transaction. Line #2 is a change record (a leading ‘c’) that alters a field (a type ‘3’ change). It alters the “name” field of element “DJJ3S#961,” which a search through the log file reveals to be the **UMLAttr** metamodel object “sz” belonging to class **HTMLList**. The record thus carries out the renaming by the left Modeler of attribute `sz` to `size`.

The mapping for the next transaction is a bit more challenging. The listed records create the `a2` association between **HTMLDoc** and **HTMLList**:

```
 1  t;YXqW0#4F1;editAssoc;1287176266254;-;;
 2  c1;;v::de.uni_paderborn.fujaba.uml.structure.UMLRole;i:YXqW0#5F1;-;i:YXqW0#4F1;
 3  c1;;v::de.uni_paderborn.fujaba.uml.structure.UMLRole;i:YXqW0#6F1;-;i:YXqW0#4F1;
 4  c1;;v::de.uni_paderborn.fujaba.uml.structure.UMLAssoc;i:YXqW0#7F1;-;i:YXqW0#4F1;
 5  c3;;i:YXqW0#5F1;name;v::doc;-;-;i:YXqW0#4F1;
 6  c3;;i:YXqW0#6F1;name;v::lists;-;-;i:YXqW0#4F1;
 7  c3;;i:YXqW0#7F1;name;v::a2;-;-;i:YXqW0#4F1;
 8  c3;;i:YXqW0#5F1;revLeftRole;i:YXqW0#7F1;-;-;i:YXqW0#4F1;
 9  c3;;i:YXqW0#7F1;leftRole;i:YXqW0#5F1;-;-;i:YXqW0#4F1;
10  c3;;i:YXqW0#6F1;revRightRole;i:YXqW0#7F1;-;-;i:YXqW0#4F1;
11  c3;;i:YXqW0#7F1;rightRole;i:YXqW0#6F1;-;-;i:YXqW0#4F1;
12  c3;;i:YXqW0#5F1;adornment;v::2;v::0;-;i:YXqW0#4F1;
13  c3;;i:YXqW0#5F1;card;i:WnHJn#491;-;-;i:YXqW0#4F1;
14  c3;;i:qyd2E#E2;roles;i:YXqW0#5F1;-;-;i:YXqW0#4F1;
15  c3;;i:YXqW0#5F1;target;i:qyd2E#E2;-;-;i:YXqW0#4F1;
16  c3;;i:YXqW0#6F1;card;i:WnHJn#591;-;-;i:YXqW0#4F1;
17  c3;;i:qyd2E#E3;roles;i:YXqW0#6F1;-;-;i:YXqW0#4F1;
18  c3;;i:YXqW0#6F1;target;i:qyd2E#E3;-;-;i:YXqW0#4F1;
19  c3;;i:YXqW0#6F1;sortedComparator;v::;-;-;i:YXqW0#4F1;
20  c3;;i:YXqW0#5F1;sortedComparator;v::;-;-;i:YXqW0#4F1;
21  c3;;i:YXqW0#6F1;sortedComparator;-;v::;-;i:YXqW0#4F1;
22  c3;;i:qyd2E#B2;elements;i:YXqW0#7F1;-;-;i:YXqW0#4F1;
23  c3;;i:YXqW0#7F1;diagrams;i:qyd2E#B2;-;-;i:YXqW0#4F1;
```

After the transaction record in line #1, there are three create object change records (type ‘1’ changes). The entities created are the Fujaba metamodel objects needed to define an association. In order, they are: a left **UMLRole**, a right **UMLRole**, and a **UMLAssoc** to link them. The records in lines #5-6 give these objects their names. Most of the remaining records go into building the structure of metamodel objects pictured on the left side of Figure 53. Intermixed with these records about the *model*, are records about the model’s *view* in Fujaba. This is a common confounding source of trouble for merge tools, as was brought out in Subsection 3.5.

To convert this operation-based transaction into its state-based normal-form, the Fujaba interface of Mirador must map the Fujaba metamodel construct of Figure 53*a*

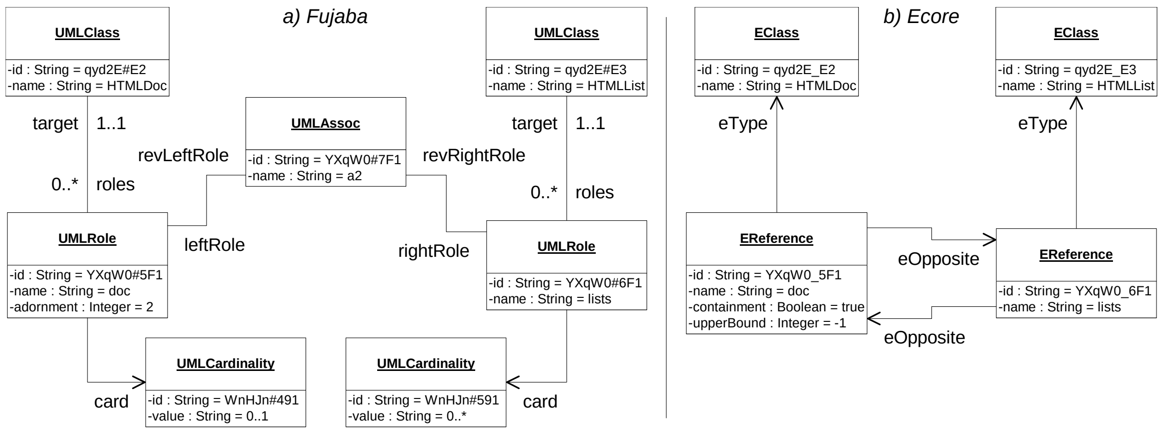


Figure 53: Mapping a Fujaba association to Ecore references

into the structure of Ecore metamodel objects in Figure 53*b*. The decoupling interface must handle many such mappings, but that of association to reference is probably the most complex. A limitation of using Ecore is that not everything *can* be mapped. One example is visibility—Ecore has no means for expressing the concept.

7.2.2 Differencing Models and the Difference Metamodel

Once the models to be merged are expressed in the normal-form, it is possible to calculate the difference of state between any two models of whatever origin. This is the Model Differencing workflow phase. Its postions here in the application is a result of GUI concerns, while its position in the workflow overview was driven by considerations of reader comprehension.

Mirador takes differences between the left model and the base model, and the right model and the common base. In a two-way merge, the base model would be empty. Regardless, the results are two difference models expressed in the extended normal-form. The extended normal-form is the Ecore metamodel (MM) augmented by a slightly updated version of the difference metamodel (MMD) proposed by Cicchetti et al. [2007]. The MMD enables capturing the differences between models as a first-class artifact.

The MMD (left side of Figure 54) stipulates the derivation of three new types for each element type of the MM. A model fragment from the running example, placed on the right side of the figure, illustrates its usage. The appearance of MMD elements in the fragment indicates that the model has been changed in some way. Specifically,

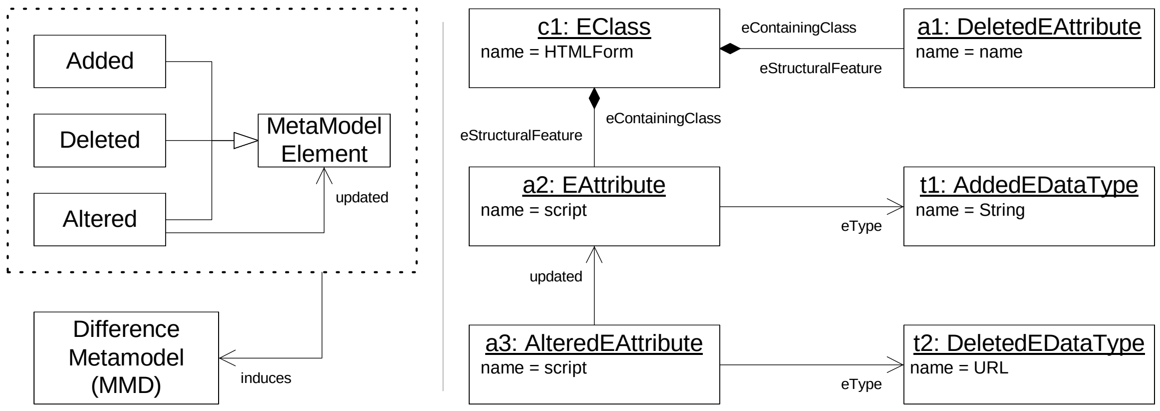


Figure 54: Difference metamodel and a differenced model fragment

class **HTMLForm** has had its **name** attribute removed and its **script** attribute's type switched from **URL** to **String**, which is shown to be a newly added data type.

In general, adding a new element **X** inserts a subclassed **AddedX** object into the difference model; deleting an element **Y** removes it, but leaves a "ghost" **DeletedY** object in the difference model; and changing an element creates an **AlteredZ** object that is a copy of **Z** *before* it is updated, with a reference back to the original, but now updated **Z**. Care must be taken regarding IDs when copying an altered element. The ID of the copy must be changed so as to distinguish it from the original, but not break its link to the original or lose it in the model. For instance, short of keeping a special list, there is no way to find object $a3$ in the model, because **Altered** elements are not referenced by any other elements. Mirador solves the problem by assigning the copied object the original ID, but with a special character appended to it. Now, given an ID to an MM object, the tool can locate its shadowy **Altered** companion should one (there is never more than one) be present.

The end result is a representation, in one artifact, of the model state *both* before and after being changed. The initial model consists of all the **Altered** and **Deleted** MMD elements ($a3, a1, t2$), and any MM elements that are not referenced by an *updated* association ($c1$), whereas the final model consists of all MM elements ($c1, a2$), plus the **Added** MMD elements ($t1$). Everything in the final model is reachable from the root node ($c1$ here), but reaching the initial model requires some sort of external reference to the **Altered** MMD elements, such as the special IDs described above.

We have made two minor modifications to the difference metamodel. First, the

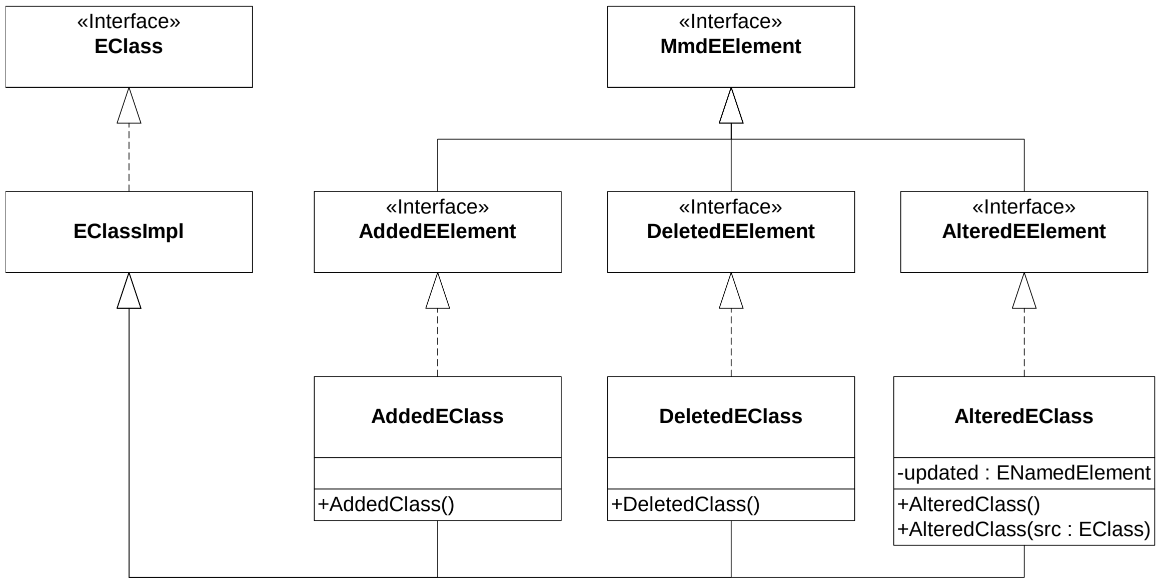


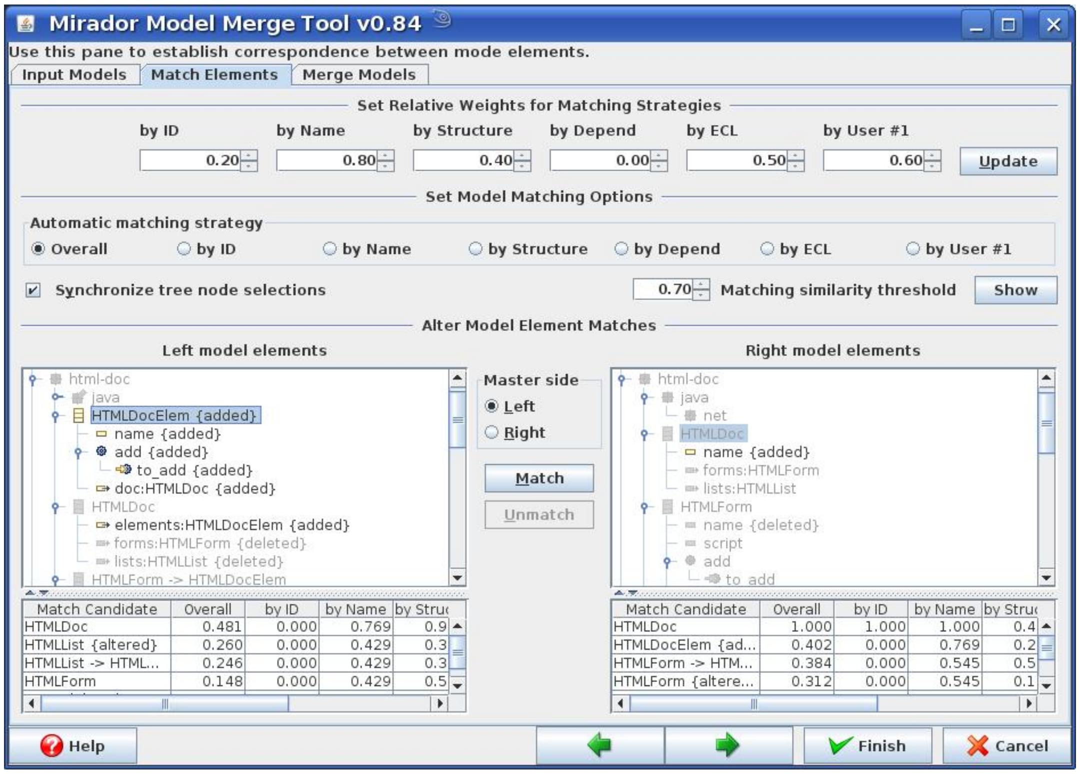
Figure 55: MMD extension of EClass and added interface hierarchy

original **Changed** MMD metaclass was renamed to **Altered** to avoid confusion with model change, change operation, etc. Second, a shallow copy constructor was added to the **Altered** classes to ease building of the difference model. There is no need to make a deep copy of an object, because either what it references has not changed or, if changed, it will get its own **Altered** copy, and so on.

The MMD extension to the **EClass** MM type is shown in Figure 55, along with a hierarchy of interfaces we have defined. The Ecore **EClass** is actually an interface realized by class **EClassImpl**, from which its three MMD classes are derived. Each of these types in turn, realize an interface specific to their function in the MMD, which are themselves derived from an overarching interface for MMD elements. The interfaces make for a cleaner and more polymorphic implementation. Every Ecore meta-element has been given a comparable structure.

7.3 Element Match Panel

Step 5 of the "Run Interactive Model Merge" use case moves the Merger from the Model Input Panel to the Element Match Panel seen in Figure 56. The left and right models are shown as trees, which, while having the virtue of simplicity, emphasize the ownership and containment relations of the models' elements. Displaying the models



Mirador Model Merge Tool v0.84

Use this pane to establish correspondence between mode elements.

Input Models | Match Elements | Merge Models

Set Relative Weights for Matching Strategies

by ID	by Name	by Structure	by Depend	by ECL	by User #1
0.20	0.80	0.40	0.00	0.50	0.60

Set Model Matching Options

Automatic matching strategy: [✓ Overall] by ID by Name by Structure by Depend by ECL by User #1

[✓] Synchronize tree node selections

0.70 Matching similarity threshold

Alter Model Element Matches

Left model elements

```
html-doc
  java
  HTMLDocElem {added}
    name {added}
    add {added}
      to_add {added}
    doc:HTMLDoc {added}
  HTMLDoc
    elements:HTMLDocElem {added}
    forms:HTMLForm {deleted}
    lists:HTMLList {deleted}
  HTMLForm -> HTMLDocElem
```

Match Candidate	Overall	by ID	by Name	by Struc
HTMLDoc	0.481	0.000	0.769	0.9
HTMLList {altered}	0.260	0.000	0.429	0.3
HTMLList -> HTML...	0.246	0.000	0.429	0.3
HTMLForm	0.148	0.000	0.429	0.5

Master side: [✓ Left] Right

Right model elements

```
html-doc
  java
    net
  HTMLDoc
    name {added}
    forms:HTMLForm
    lists:HTMLList
  HTMLForm
    name {deleted}
    script
    add
      to add
```

Match Candidate	Overall	by ID	by Name	by Struc
HTMLDoc	1.000	1.000	1.000	0.4
HTMLDocElem {ad...	0.402	0.000	0.769	0.2
HTMLForm -> HTM...	0.384	0.000	0.545	0.5
HTMLForm {altere...	0.312	0.000	0.545	0.1

Figure 56: Mirador Element Match Panel

as trees, also avoids conflating changes to the model with changes to its view.

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 a tooltip with more information. Selecting an element in a tree will cause to be displayed in the table directly beneath that tree all of the element's possible matches (i.e., meta-elements of the same type) found in the other tree. Each match candidate appears in the table with measures of its similarity to the selected element—one measure for each loaded matching strategy—and is ranked according to its score for the selected "Automatic matching strategy."

Up to seven strategies may be loaded, with the GUI dynamically adapting to accommodate the necessary controls. Two of the strategies may be user-supplied Java classes, and one a script written in the Epsilon Comparison Language (ECL) [Kolovos, 2009]. The "by ID" strategy, however, is always loaded. The values of a candidate's individual strategy measures of similarity are used to compute an overall similarity score for the candidate. The computation is a weighted distance function, in which the weight for each strategy is set through the controls running across the top of the Element Matching Panel. Putting a strategy's weight to zero effectively removes it from consideration in the calculation.

The tables and trees may be synchronized so that selecting one of a paired element in either tree (signified by grayed out nodes) will highlight its partner in the other tree. Clicking on a candidate match in a table will highlight the referenced element in the other table's tree regardless of its paired status. Available elements may be paired by selecting them and pressing the **Match** button. Likewise, a matched pair may be broken apart by pressing **Unmatch**.

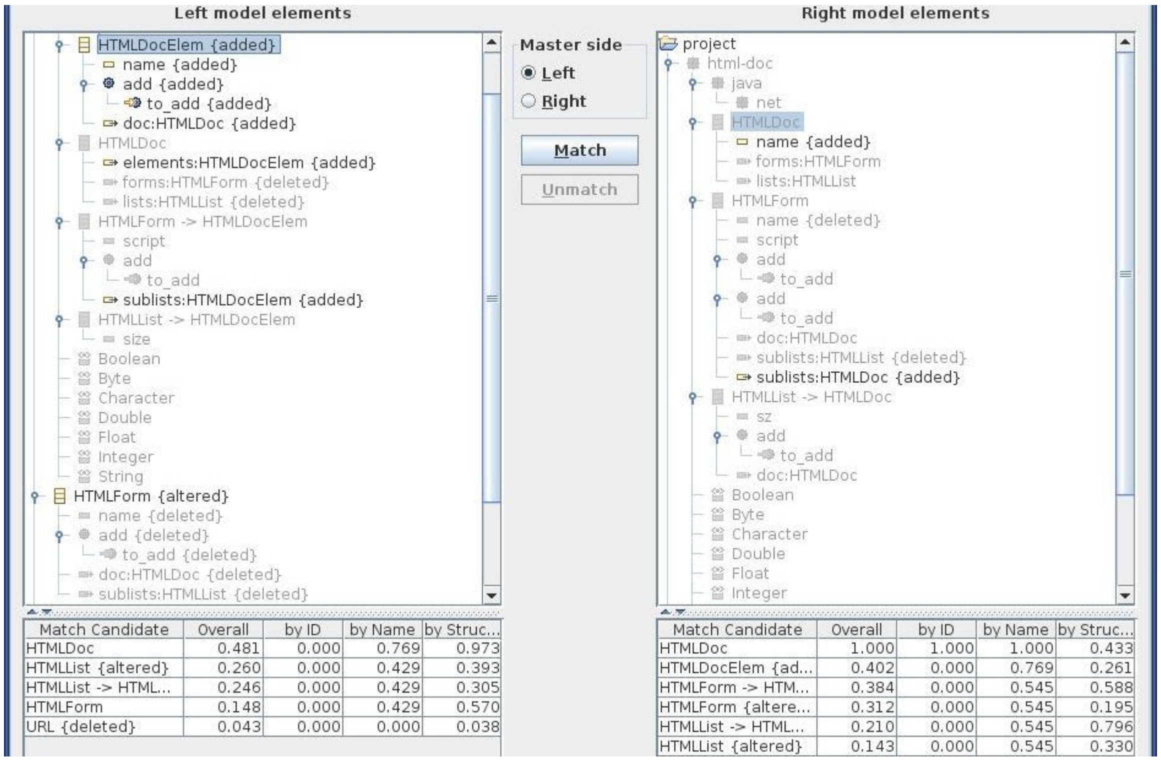
7.3.1 Comparing Models and Matching Elements

The model elements are matched upon panel entry based on their overall similarity scores. An element is matched with its highest ranked *available* candidate that is at, or above, the value in the "Matching similarity threshold" field. The Merger may change the comparison method used for this purpose with the "Automatic matching strategy" radio buttons, and can also alter the threshold value or strategy weights used in the calculations. Changes to any of these controls will trigger a rematching, which will respect any manual pairing or unpairing the Merger may have already done in the session.

In the event of a matching ambiguity, the setting of the "Master side" radio buttons comes into play. It specifies which model's measure to use in those cases where the similarities between two matchable elements are not symmetrical, i.e., the left-to-right value is different than the right-to-left value. It is also used in merging to resolve those conflicts in which the changes are in contradiction, because they do the same thing, but only one needs to be executed.

The enlarged cutaway of Figure 57 gives a clearer view of the model trees and the match candidates for the two highlighted tree elements. For this screenshot, the only strategies assigned non-zero weights are "by ID" (0.2), "by Name" (0.7), and "by Structure" (0.4). Accordingly, they are the only components that figure in calculating the overall measures.

Most of the trees' elements have been matched, as signified by their being grayed out. Based on the overall measure—selected as the automatic matching strategy—the



```
HTMLDocElem {added}
    name {added}
    add {added}
        to_add {added}
    doc:HTMLDoc {added}
HTMLDoc
    elements:HTMLDocElem {added}
    forms:HTMLForm {deleted}
    lists:HTMLList {deleted}
HTMLForm -> HTMLDocElem
    script
    add
        to_add
    sublists:HTMLDocElem {added}
HTMLList -> HTMLDocElem
    size
Boolean
Byte
Character
Double
Float
Integer
String
HTMLForm {altered}
    name {deleted}
    add {deleted}
        to_add {deleted}
    doc:HTMLDoc {deleted}
    sublists:HTMLList {deleted}
```

Match Candidate	Overall	by ID	by Name	by Struc...
HTMLDoc	0.481	0.000	0.769	0.973
HTMLList {altered}	0.260	0.000	0.429	0.393
HTMLList -> HTML...	0.246	0.000	0.429	0.305
HTMLForm	0.148	0.000	0.429	0.570
URL {deleted}	0.043	0.000	0.000	0.038

```
project
  html-doc
    java
      net
    HTMLDoc
      name {added}
      forms:HTMLForm
      lists:HTMLList
    HTMLForm
      name {deleted}
      script
      add
        to_add
      add
        to_add
      doc:HTMLDoc
      sublists:HTMLList {deleted}
      sublists:HTMLDoc {added}
    HTMLList -> HTMLDoc
      sz
      add
        to_add
      doc:HTMLDoc
    Boolean
    Byte
    Character
    Double
    Float
    Integer
```

Match Candidate	Overall	by ID	by Name	by Struc...
HTMLDoc	1.000	1.000	1.000	0.433
HTMLDocElem {ad...	0.402	0.000	0.769	0.261
HTMLForm -> HTM...	0.384	0.000	0.545	0.588
HTMLForm {altere...	0.312	0.000	0.545	0.195
HTMLList -> HTML...	0.210	0.000	0.545	0.796
HTMLList {altered}	0.143	0.000	0.545	0.330

Figure 57: Element trees and candidate similarity tables for matched models

best match for **HTMLDocElem** is **HTMLDoc**, but its overall score is too low to meet the specified threshold of 0.6. The "by name" strategy, however, *would* meet the set threshold had it been picked as the automatic strategy. As it is, not sharing the same ID has a weighted downward effect on overall similarity. The converse is not true: if the IDs in a comparison are identical, the "by ID" value is passed straight through, disregarding any weights. This is because matching on ID is unlikely to raise false positives.

In the implementation, the Model Differencing phase comes before the Model Comparison phase, and for this reason the trees of the panel display *difference* models, hence the presence of MMD metaobjects in the trees. Matching is not affected by whether model elements are MM or MMD meta-elements; either can be matched. For instance, an MM element on one side may be matched with a {*deleted*} MMD element on the other side.

The MMD elements reflect the evolution that the base model replicas have undergone. For example, on the left side, the **HTMLDocElem** class has been {*added*} to the base, as have its members **name**, **add** and **to_add**, as well as its reference **doc**, the opposite end of which has been {*added*} to **HTMLDoc** as **elements**. The **name** attribute that is pulled-up into the new class is no longer found in **HTMLForm**, but instead appears in the {*altered*} shadow class at the bottom of the tree as having been {*deleted*}. The shadow class captures **HTMLForm** before it is updated to subclass **HTMLDocElem**.

There are two means of extending Mirador's element matching capabilities with custom strategies. The easier way is to write a script in the Epsilon Comparison Language to define rules by which model elements may be matched. The simple (and somewhat pointless) script below has two rules: one for comparing packages and another for comparing classes.

```
rule Match_Packages
  match l : Left!EPackage with r : Right!EPackage {
    compare : true
    do {
      matchInfo.put("mirador", 0.6);
    }
  }
rule Match_Class_Names
  match l : Left!EClass with r : Right!EClass {
    compare : l.name = r.name
    do {
      matchInfo.put("mirador", 0.7);
    }
  }
```

The first rule, `Match_Packages`, matches any package of the left model with any package of the right model and returns a similarity measure of 0.6 through the specially defined `mirador` variable. Rule `Match_Class_Names` is a little more discriminating in that it actually tests to see if the class names are identical, returning 0.7 if they are. For Mirador to find an ECL script, it must be named "mirador-evaluator.ecl" and be located in a certain directory.

The more involved customization route is to create a Java class to be loaded as one of the "by User" strategies. This requires creating a subclass of the abstract **SimilarityEvaluator** class and overloading its **evaluate** method. The method returns a normalized value that reflects the similarity between its two model element parameters. As an example, the code for the **evaluate** method of class **IdEvaluator**, a particularly simple evaluator, is reproduced below.

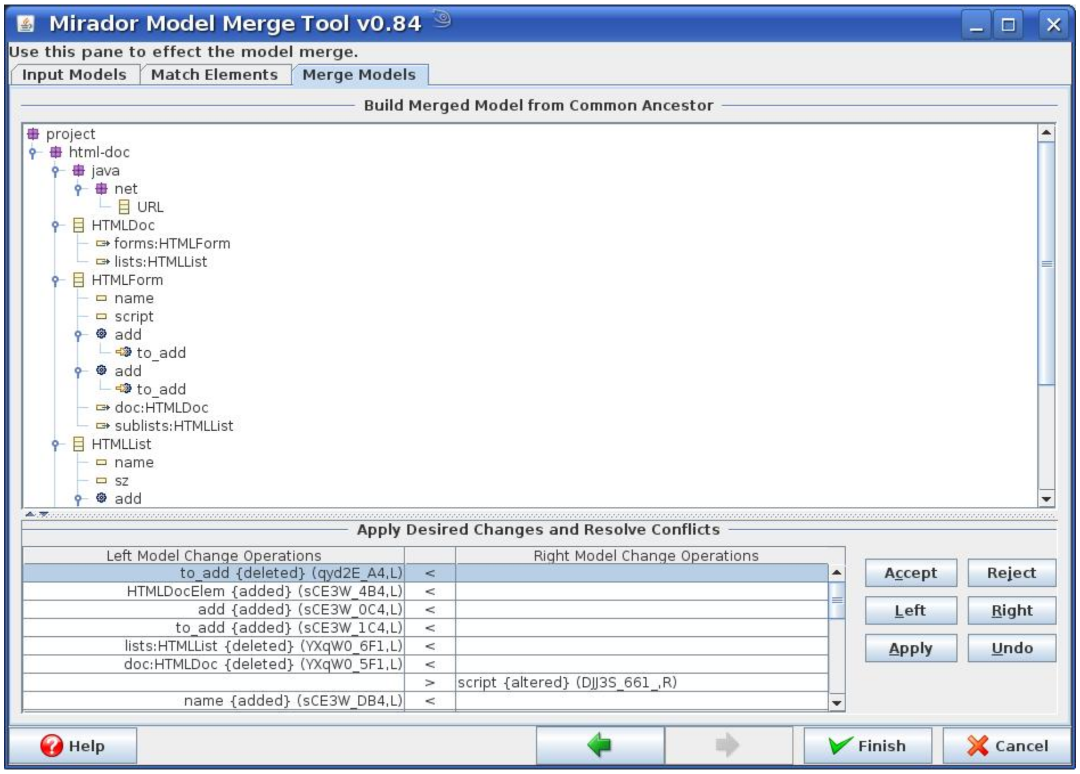
```
/**
 * Evaluates similarity of IDs from one model element to another. As measures
 * are not always symmetrical, a direction of evaluation is implied by the
 * arguments. The measure is a normalized representation of similarity.
 *
 * @param  from_element  Model element from which to measure.
 * @param  to_element  Model element to which to measure.
 * @return  Normalized similarity: 0.0 = no similarity, 1.0 = identical
 */
@Override public float evaluate(EcoreExtra from_element, EcoreExtra to_element) {
  float similarity;
  // No gray area: either the IDs are identical, or they are not.
  if (from_element.getId().equals(to_element.getId()))
    similarity = 1.0f;  // Same IDs; created in base model.
  else
    similarity = 0.0f;  // Different IDs.

  return similarity;
}
```

7.4 Model Merge Panel

The Model Merge Panel is entered at step 8 of the "Run Interactive Model Merge" use case and is shown in Figure 58. The panel is divided into two sections. The upper portion displays the current state of the merged model using the same tree structure as in the Match Elements Panel. In this panel section, user interactions are limited to getting model element details via tooltips, and expanding or collapsing tree nodes. The lower section is used to direct the merge: it presents the change operations from both sides of the merge and any conflicts there might be between them, knitted into a single list and placed in a table. The Merger then uses the button group to the right of the table to confirm or modify the actions proposed by this *merge plan*.

On panel entry, a **ChangeOpPlane** is constructed. Once built, this control object orchestrates the tasks remaining to complete the merge, including: building decision tables according to their default or loaded definitions, extracting change


Mirador Model Merge Tool v0.84
Use this pane to effect the model merge.
Input Models
Match Elements
Merge Models
Build Merged Model from Common Ancestor
Apply Desired Changes and Resolve Conflicts
Accept
Reject
Left
Right
Apply
Undo
Help
Finish
Cancel

```
project
  html-doc
    java
      net
        URL
    HTMLDoc
      forms:HTMLForm
      lists:HTMLList
    HTMLForm
      name
      script
      add
        to_add
      add
        to_add
      doc:HTMLDoc
      sublists:HTMLList
    HTMLList
      name
      sz
      add
```

Left Model Change Operations		Right Model Change Operations
to_add {deleted} (qyd2E_A4,L)	<	
HTMLDocElem {added} (sCE3W_4B4,L)	<	
add {added} (sCE3W_0C4,L)	<	
to_add {added} (sCE3W_1C4,L)	<	
lists:HTMLList {deleted} (YXqW0_6F1,L)	<	
doc:HTMLDoc {deleted} (YXqW0_5F1,L)	<	
	>	script {altered} (DJJ3S_661_,R)
name {added} (sCE3W_DB4,L)	<	

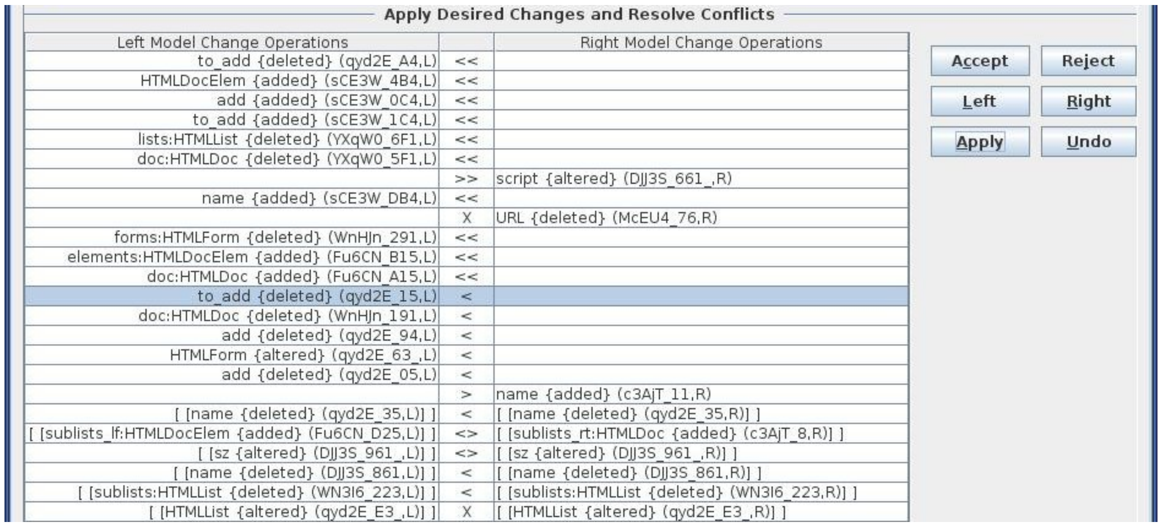
Figure 58: Mirador Model Merge Panel

operations from the matched difference models of the preceding Element Match Panel, partitioning and ordering the operations, and detecting and automatically resolving conflicts. Before examining some of these behind-the-scenes activities, usage of the panel is first explained.

Pressing the **Finish** button will cause the merged model to be patched with any unexecuted changes that have been marked as accepted (`<` or `>`). This excludes any changes that have been rejected or are involved in an unresolved conflict (`X`), and those that could not be applied due to an inconsistency error (`!`). At this point, assuming no Merger choices to the contrary, as many of the changes from both Modelers as possible have been incorporated into the merged model. It is now ready to undergo the denormalizing transformation, and to be saved in its original format(s).

7.4.1 Making Merge Decisions

When the panel first opens, its upper section is filled with a copy of the base model. Since the base model is common to both replica-versions, this copy is, in fact, the



Left Model Change Operations		Right Model Change Operations
to_add {deleted} (qyd2E_A4,L)	<<	
HTMLDocElem {added} (sCE3W_4B4,L)	<<	
add {added} (sCE3W_0C4,L)	<<	
to_add {added} (sCE3W_1C4,L)	<<	
lists:HTMLList {deleted} (YXqW0_6F1,L)	<<	
doc:HTMLDoc {deleted} (YXqW0_5F1,L)	<<	
	>>	script {altered} (DJJ3S_661_,R)
name {added} (sCE3W_DB4,L)	<<	
	X	URL {deleted} (McEU4_76,R)
forms:HTMLForm {deleted} (WnHJn_291,L)	<<	
elements:HTMLDocElem {added} (Fu6CN_B15,L)	<<	
doc:HTMLDoc {added} (Fu6CN_A15,L)	<<	
to_add {deleted} (qyd2E_15,L)	<	
doc:HTMLDoc {deleted} (WnHJn_191,L)	<	
add {deleted} (qyd2E_94,L)	<	
HTMLForm {altered} (qyd2E_63_,L)	<	
add {deleted} (qyd2E_05,L)	<	
	>	name {added} (c3AjT_11,R)
[[name {deleted} (qyd2E_35,L)]]	<	[[name {deleted} (qyd2E_35,R)]]
[[sublists_lf:HTMLDocElem {added} (Fu6CN_D25,L)]]	<>	[[sublists_rt:HTMLDoc {added} (c3AjT_8,R)]]
[[sz {altered} (DJJ3S_961_,L)]]	<>	[[sz {altered} (DJJ3S_961_,R)]]
[[name {deleted} (DJJ3S_861,L)]]	<	[[name {deleted} (DJJ3S_861,R)]]
[[sublists:HTMLList {deleted} (WN3I6_223,L)]]	<	[[sublists:HTMLList {deleted} (WN3I6_223,R)]]
[[HTMLList {altered} (qyd2E_E3_,L)]]	X	[[HTMLList {altered} (qyd2E_E3_,R)]]

Figure 59: Partially executed merge plan of proposed actions for running example

merge of the two models, *before* any Modeler changes have been made. Their changes, both conflicting and otherwise, are arranged into a sequence according to intra- (i.e., same version) change and inter- (i.e., cross version) change dependencies, with conflicting change-pairs being pushed as far to the end as possible. Put into a table, this list becomes the merge plan, which is then presented to the Merger for final approval. Figure 59 is a close-up of the merge plan for the running example.

Pending Modeler changes are listed in the left and right columns of the merge plan, respectively. The action Mirador proposes to take with regard to each change operation is put in the middle column. The Merger can override the tool's decision by highlighting a change and pressing one of the control buttons: "Accept" or "Reject" for non-conflicting changes, and "Left", "Right" or "Reject" for conflicting ones. "Apply" will execute all changes up to and including the highlighted one, in accordance with the proposed action.

Application of a change (i.e., a model transformation) patches the current version of the merged model and updates its display to reflect any visible alterations. The symbol of the change's proposed action is in turn updated to indicate the outcome of the execution. Execution stops if either an error or an unresolved conflict is encountered. It is possible to skip over these and apply any remaining changes, but there is an increased likelihood of inconsistency errors. The symbols for action proposal and execution status that may appear in the middle column are summarized here:

```
Single and conflicting changes
    <   (>)  - execute left (right) change
    << (>>)  - left (right) change executed
      !      - error executing change
Single changes
      X      - change rejected
Conflicting changes
     < >     - execute both changes
    << >>    - both changes executed
      X      - unresolved conflict
```

Executing both operations of a conflict can be done, but only if semantics are take into account. For instance, if one side renames an attribute and the other makes it static, there is no real conflict, even if one is detected, so both changes should be executed. If automatic resolution marks both changes for application, it indicates that conflict detection and conflict resolution rules are not operating at the same level. In other words, the detection rules see a conflict because they are not looking as deeply as the resolution rules. This difference in level, however, might be by design: flagging a conflict, and then automatically resolving it brings the situation to the attention of the Merger, who can then reverse the merge decision if desired.

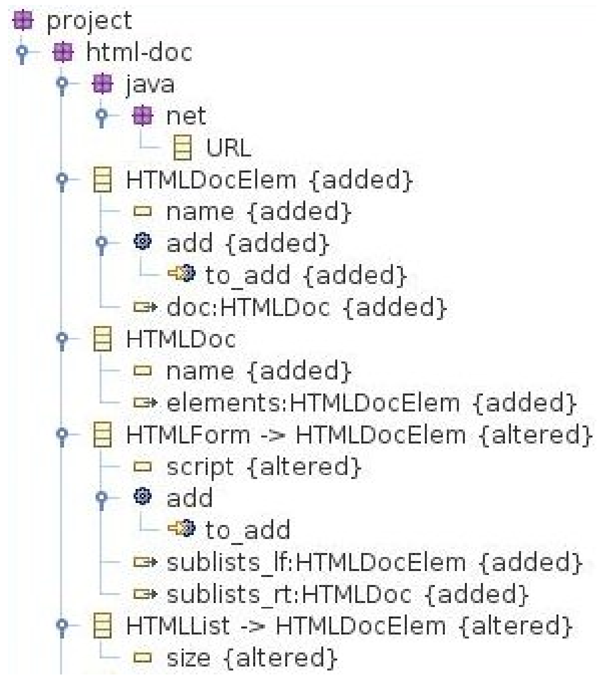
The status markers in Figure 59 indicate that about half of the change operations of the merge plan have been executed successfully. While the presentation is primitive, it is still possible to trace the application of a change to its effect on the merged model shown in Figure 60. From top to bottom, the applied changes of the plan (with '`L`' and '`R`' signifying the side) yield these effects on the model:

1. Change *L, to_add* {*deleted*} removes the parameter from one of the `add` methods of class **HTMLForm**.

2. *L, HTMLDocElem* {*added*} adds a new class to the merged model.

3. Changes *L, add* {*added*} and *L, to_add* {*added*} insert the so-named method and parameter into the new **HTMLDocElem** class, in that order.

4. *L, lists:HTMLList* {*deleted*} and *L, doc:HTMLDoc* {*deleted*} remove both sides of bidirectional association `a2`.



```
project
  html-doc
    java
      net
        URL
    HTMLDocElem {added}
      name {added}
      add {added}
        to_add {added}
      doc:HTMLDoc {added}
    HTMLDoc
      name {added}
      elements:HTMLDocElem {added}
    HTMLForm -> HTMLDocElem {altered}
      script {altered}
      add
        to_add
      sublists_lf:HTMLDocElem {added}
      sublists_rt:HTMLDoc {added}
    HTMLList -> HTMLDocElem {altered}
      size {altered}
```

Figure 60: Merged running example after execution of 11 change operations

5. *R, script* {*altered*} replaces the attribute's data type of **URL** with **String**.

6. *L, name* {*added*} pulls-up subclass attribute `name` into **HTMLDocElem**.

7. Since it is rejected, change *R, URL* {*deleted*} has no effect on the merge.

8. *L, forms:HTMLForm* {*deleted*} takes out half of the `a1` link. The other half, *L, doc:HTMLDoc* {*deleted*}, is a few changes lower in the plan.

9. Bidirectional association `a4` is put into the merged model as two references with *L, elements:HTMLDocElem* {*added*} and *L, doc:HTMLDoc* {*added*}.

The sequence of changes in a merge plan is only partially ordered. In some cases, the reasons for the ordering are evident (e.g., creation of **HTMLDocElem** before its `name` attribute), but in others it is arbitrary (e.g., lower bound of `sz` being set to 1 before being renamed). It is this relaxed ordering that lets Mirador move conflicts towards the bottom of the list. The benefit is that the Merger can study a nearly fully-formed merged model before having to make any decisions regarding conflict resolution.

Six conflicts appear at the end of the merge plan in Figure 59. Two have to do with the `name` attributes of **HTMLForm** and **HTMLList** being deleted in order to pull them up into **HTMLDocElem** on the left and **HTMLDoc** on the right. The resolution rules recognize that the conflict concerns operation preconditions (the second delete will have nothing to operate on), but also that executing only one of the deletes will satisfy both changes without error. Consequently, the rules choose to apply the delete operation of the conflicting pair that comes from the "master side" of the merge (Subsection 7.3.1). This satisfies the intent of both changes, eliminates the conflict, and does not corrupt the model. The removal of the unidirectional `a3` association on both sides causes another delete-delete conflict over its `sublists` reference, which is handled in the same way.

There is one add-add conflict: a unidirectional association from **HTMLForm** targeting a different class on each side, but with the same role name. This is certainly a problem for code generation, but not so much for modeling things are not so clear cut. If the associations are matched then it makes sense to consider the changes as mutually exclusive and force a choice. If however, as in this case, they are not matched, a more prudent course, as followed here, is to give the roles different names and apply both changes. Of course, the modeler is free to reject this recommendation.

The changes being considered for attribute `sz` are actually non-overlapping, so the proposal is to execute both. On the left, the name is being changed to `size`, while on the right its lower limit is being set. As already mentioned, the nature of these changes could have been determined during conflict detection and a conflict not raised, but by flagging it and then automatically resolving it, the changes become visually linked, clueing the Merger in to the changes, but not insisting on any action on their part.

In the last conflict, class **HTMLList** is changed on the left to be derived from **HTMLDocElem**, and on the right from **HTMLDoc**. Since no rules exist to decide this situation, the conflict is marked as unresolved, and it is up to the Merger to make the decision. Due to the large proportion of changes that have already been applied by the time this point is reached, the Merger should have a relatively easy time of making a call. For instance, knowing that **HTMLDocElem** is now part of the model makes inheriting from it, rather than from **HTMLDoc**, a sensible choice.

We have been concentrating on the interactive path through the activity diagram of Figure 45. But the design documents another path, what Bendix et al. [2010] refer to as batch merging. Both paths hit all phases of the hybrid workflow to perform a merge, but batch mode bypasses the user input control interfaces of the comparison and resolution phases. This allows Mirador to be used *sans* GUI, and still produce a merged model and textual merge report.

The hallmark of batch mode is its ability to provide rapid feedback with little user input, and its amenability to automation (e.g., scripts, job scheduling, text analysis programs, etc.). Since decisions with regard to element matching and conflict resolution are made by the tool instead of a human Merger, as in interactive mode, unresolved conflicts may leave the outcome not fully merged.

Whereas interactive mode may accept arguments or options from the command line, or through a configuration file, for batch mode passing data in this manner is mandatory. For example, assuming that the class path is properly set up, starting the running example in batch mode from the command line would look like this:

```
java ca.dsrg.mirador.Mirador -b -c=mirador.cfg
```

The “mirador.cfg” configuration file might contain something along these lines:

```
-d -mi=0.2 -mn=0.7 -ms=0.4 -me -u1=UserEvaluator1 -mt=0.6  // Options
-td=usr/table-before.ddf -tr=usr/table-resolve.ddf         // Table definitions
html/base.ctr  html/left.ctr html/right.ctr                // Model files
```

Details about the Mirador invocation syntax and the format of its configuration file are provided in Appendix C.

7.4.2 Merging on the Change Plane

The topic of this subsection is the algorithm for carrying out the actual merging of model change operations, namely, the Operation Extraction, Conflict Detection, and Conflict Resolution phases of the hybrid workflow, but without the manual intervention discussed in the preceding subsection. The conceptual devices introduced in Chapter 4 find realization in the algorithm’s code: the conflict matrix takes the form of a collective of relation maps, one for each change operation; and the change plane, as mentioned at the beginning of this section, is instantiated as an object of type **ChangeOpPlane**.

Being the platform upon which the final sequence of change operations is decided, the change plane is central to Mirador's operation-based merging. As such, its incarnation in the code is the driver of the algorithm, which is divided into five stages. Each stage is invoked by one of the private methods of the change plane seen in Figure 61 and described in the correspondingly titled paragraphs that follow.

Before starting the merge, **ChangeOpPlane** must build two **DecisionTable** objects on which it depends: a `before_table`, which itself is composed of two linked tables, and a `resolve_table`. The tables are constructed either from the default definitions contained in the class itself or from external definition files loaded with the Model Input panel or configuration options.

Operation Extraction

To replace state-based artifacts with operation-based ones requires knowing which change operations were executed in modifying the replicas. Since this information is destroyed by the state-based front-end of the hybrid workflow, the operations must be extracted from the two difference models outputted by the Model Differencing phase. The `extractAtomicChangeOps` method of **ChangeOpPlane** is used for this purpose. Each operation extracted is initially recreated as one of three types of **AtomicChangeOp**, which itself is an abstract subtype of class **ChangeOp**. The other subtypes are covered in the algorithm stages that use them.

The subtypes of **AtomicChangeOp** correspond exactly to the similarly named metaclasses of the difference metamodel, with a single object being constructed for each such metamodel element found in either of the difference models. Each atomic change object is linked to an Ecore **ENamedElement** that is the `target` of the change. The referenced element may be one of several enumerated **EcoreType** tags: `PACKAGE`, `CLASS`, `ATTRIBUTE`, `OPERATION`, `PARAMETER`, `REFERENCE`, or `DATATYPE`. If a change happens to be of type **AlterChangeOp**, then it will also have an `update` reference to the model element it alters. All this information may be directly obtained from the difference model.

The outcomes from the stage are two lists of atomic change operations. The list orders are a byproduct of the way the graphs of the difference models are traversed and bear no resemblance to the original sequence of model changes. The place a

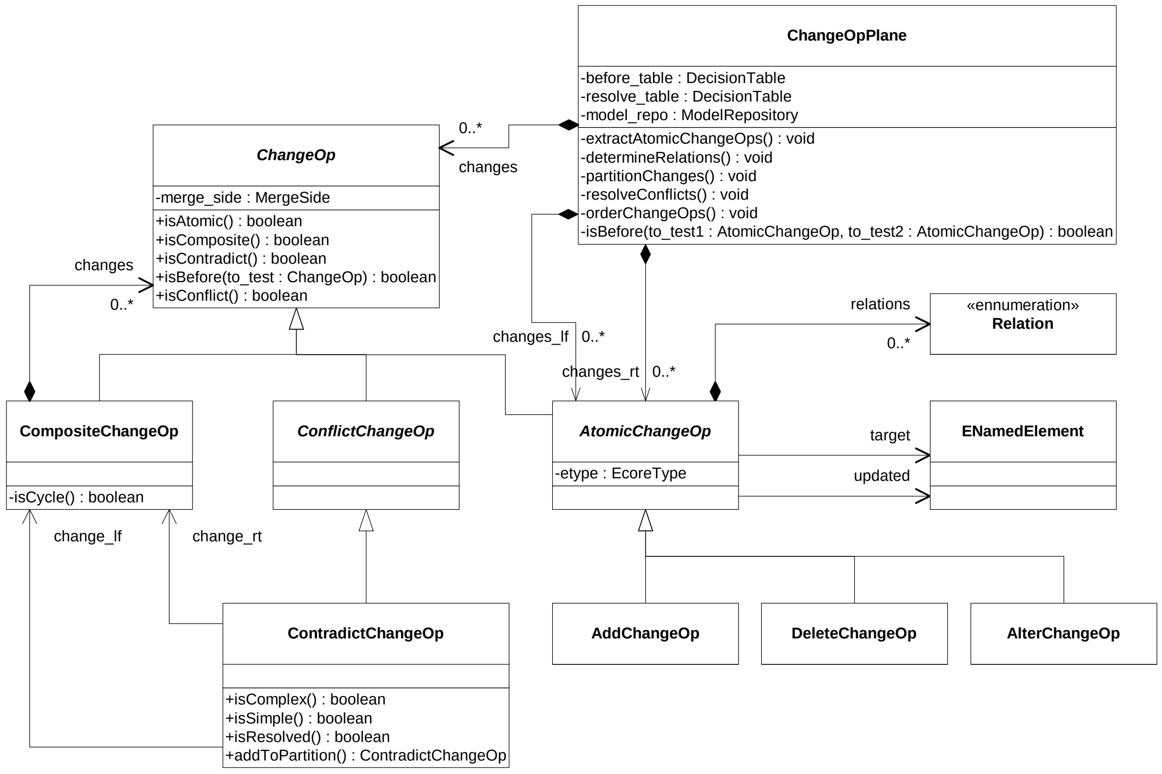


Figure 61: Change operation class hierarchy

change will come to occupy in its list, and eventually in the merged operational trace of both lists, is determined by examining its relationships with all the other changes.

Determine Change Relations

Every **AtomicChangeOp** tracks how it relates to the others in its own `relations` map. The key for a particular atomic change's map is the other changes it has a relationship with. The value of each map entry is the nature of the relationship, expressed as an enumerated **Relation** tag of either `BEFORE`, or `CONFLICT`. Changes with which the map owner does not have one of these kinds of relationships will have no entry in the map.

The `determineRelations` method uses the *before* predicate to establish the positional preferences of change operations, as explained in Subsection 4.3.1. The predicate is implemented as method `isBefore` in class **ChangeOpPlane**. The method tests the relative ordering of two atomic changes, returning *true* if the first claims positional preference over the second. When testing a pair of changes, `isBefore`

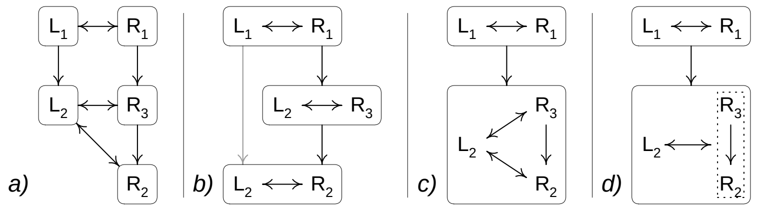


Figure 62: *a)* Change dependencies in blocks, *b)* non-exclusive conflicts, *c)* exclusive conflicts using a composite block, *d)* with conflicts collapsed

makes use of `before_table` to determine the outcome. The table is broken into two sub-tables: the first, `before_same`, is used to evaluate operations from the same side of the merge, while the second, `before_cross`, is used to evaluate operations from opposite sides of the merge.

Cross-side testing discovers a conflict when `isBefore` returns *true* for a pair of operands no matter their order. The identification of all change operation relations essentially creates a conflict matrix that is distributed across all the changes, rather than kept in a single construct. By consulting this map, each **AtomicChangeOp** is able to tell where it stands in relation to every other **AtomicChangeOp**, on either side of the merge. **ChangeOp**.`isBefore` accesses the status of the test for a given change; it does not actually conduct the test.

Partition Change Operations

The intent of method `partitionChanges` is to isolate the effects of conflicts on the other changes. It does this by putting contradicting operations into mutually exclusive blocks, as described in Subsection 4.3.2. Figure 62 reproduces the change cycle example from that section. In part *a*, all of the model changes, which start the process as **AtomicChangeOp** objects, are partitioned into their own blocks. Each block is a **CompositeChangeOp**, which may be simple and contain only a single atomic change, as in this case, or complex and contain other composites.

In Figure 62*b*, the conflicting pairs of composite objects are put into **Contradict-ChangeOp** objects, and referenced through the `change_lf` and `change_rt` members. The lower two contradicting change blocks, because they are not mutually exclusive, are composed again in Figure 62*c* into another composite change object. This composite block has a cycle, which `isCycle` is able to detect, and which is broken in part *d* by collapsing its conflicts. Below is the portion of the merge report outputted by this stage when merging the changes of Figure 62. The curly braces indicate a **ContradictChangeOp**, square brackets, a **CompositeChangeOp**, and an alphanumeric label, an **AtomicChangeOp**.

```
    --- CHANGE OP PARTITIONS ---
1  { [ [L1] ], [ [R1] ] }
2  { [ [L2] ], [ [R3], [R2] ] }
```

Line #1 is a textual rendering of the upper block of Figure 62*d*, with its constituents occupying the left and right composite changes of a contradicting change. The components of the figure's lower block are in line #2, where atomic change L_2 is wrapped in a composite change, which in turn is inside the left composite change of a contradicting change. Meanwhile, the atomic changes of the R_3–R_2 dependency chain are also wrapped in composite changes, and grouped into the right composite change of the conflict. The lists of extracted operations have now been merged into a single list of atomic, composite, and conflicting change operations having no particular order.

Resolve Conflicts

The table-driven `resolveConflicts` method automatically reconciles any conflicts that match one of the patterns defined by the rules of `resolve_table`. The conflicts of the merged list received from the previous stage are evaluated one by one against this table. Each evaluation returns an **ActionSeq** object—a sequence of **TableAction** objects, which may be empty. If non-empty, each action of the sequences is executed, and the conflict resolved.

Execution only selects which composite change of a conflict is to be applied to the model; it does not apply the change. The enumerated **MergeSide** choices for resolution are: `NONE`, `BASE`, `LEFT`, `RIGHT` and `BOTH`. The first signifies that the conflict was not resolved, the second, that both changes of the conflict are to be ignored, the next two, that the conflict has been resolved to the indicated side, and the last, that both changes of the conflict are to be applied. In all but the first case, the `is_resolved` flag of the conflict is set to *true*.

The **DecisionTable** class contains definitions for four variations of abstract class **TableAction**: `do_left`, `do_right`, `do_master`, and `do_rename`, which override the `do_action` method to make their respective resolution choices. The first two make the obvious choices, while an **ActionSeq** that contains the two of them will make the `BOTH` choice. The third class runs one of the `do_action` methods of the other two, depending on how the "Master side" radio buttons of the Element Match Panel have been set. The last appends a "_lf" to the target of the left change and a "_rt" to that of the right change. This allows Mirador to circumvent the duplicate element problem, without discarding a potentially important change.

Order Change Operations

The last stage of the algorithm is almost anticlimactic. Since the changes have already been merged into a list of operations, and their conflicts resolved to the extent possible, the merge is, in a sense, a *fait accompli*. Nonetheless, the function of `orderChangeOps` is vital to the outcome of the merge. It ensures that the merged changes will not be used to patch the base model in an order that would violate any of their dependencies. It also pushes out the frontier set, delaying the requirement for the manual resolution of any remaining conflicts as long as possible.

A final ordering is done based on the relation map kept by each change. Taken as a whole, the maps establish a graph of dependencies between the changes of the list: any circular dependencies in the graph will have already been broken in the partitioning stage, and any freedom found in the graph is exploited to move conflicts as far down the list as possible, with the unresolved ones placed below the resolved ones. Once detected, even if resolved, a conflicting pair of changes is kept together, both to facilitate traceability and to allow overriding of the resolution choice.

The model merge is now complete. All that remains is to patch the base model with the transformation outputted by the algorithm. In batch mode, the Model Patching workflow phase would be executed directly, but in interactive mode, the Merger is given the opportunity to examine, and possibly alter, the results of the automatic resolution.

7.4.3 Detecting and Resolving Conflicts with Tables

Decision tables provide an overview of control flow that is easier to understand and review than code [Pooch, 1974], making for flexible and rapid program changes. In the Determine Change Relations stage of the merge algorithm, tables direct dependency discovery and conflict detection (i.e., *before* predicate evaluation), and for the Resolve Conflicts stage, a table decides which side of a contradicting change operation to execute. Three distinct user types are supported: the turnkey user who relies on the built-in tables; the motivated user who tweaks table rules; and the practitioner of "domain specific merging" who replaces the tables entirely with their own set.

Table Specification

Rather than use its built-in tables, Mirador can load user-specified tables from files. One such file is shown in Figure 63. The main components are: conditions, actions, and rules. The names of the `[Conditions]` section correspond to functions to be evaluated, and those of the `[Actions]` section to procedures to be executed. `[Table]` simply provides a name and delimits sub-tables in the same file.

Each column of desired condition outcomes forms a rule. Rules are evaluated from left to right, and condition results from top to bottom. Evaluation ends when a pattern is matched, or the rules have been exhausted. Conditions constitute the table's domain, restricting these to predicates gives a value range of *true* ('`Y`'), *false* ('`N`'), or *don't care* ('`-`'). This makes the tables *limited-entry types*, meaning that the *rule mask technique* can be used for pattern matching [King, 1966].

A rule selects a condition for testing by placing a '`Y`' or an '`N`' opposite the condition's name. Conditions with a '`-`' are ignored. To satisfy a rule, all of its tested conditions must return the expected value. Once satisfied, a rule fires, executing the tagged procedures of the `[Actions]` section in the column directly beneath the rule, in numerical order. Actions not to be executed are marked with a '`0`'.

Conflict Detection and the *before* Predicate

When applied to two changes, the *before* predicate of Lippe and van Oosterom [1992] imposes an ordering if one operation must come before the other. While adequate in preventing inconsistencies among operations on the same replica, *before* is lacking

```
[Table]
before_same   // Evaluates before(op1, op2) for same side change operations.
              //        +-------------- Element is added to a container that itself is added.
              //        |  +----------- Added class element is target of an added reference.
              //        |  |  +-------- Element is deleted from a container that is deleted.
              //        |  |  |  +----- Altered class subtypes added class.
              //        |  |  |  |  +-- Catch-all rule, invokes chained decision table.
              //        |  |  |  |  |
[Conditions]  // Rule#  1  2  3  4  5
on_same_side            -  -  -  -  N
op1_is_add              Y  Y  -  Y  -
op1_is_delete           -  -  Y  -  -
op1_is_alter            -  -  -  -  -
op1_is_class            -  Y  -  Y  -
op2_is_add              Y  Y  -  -  -
op2_is_delete           -  -  Y  -  -
op2_is_alter            -  -  -  Y  -
op2_is_reference        -  Y  -  -  -
op2_is_class            -  -  -  Y  -
op1_contains_op2        Y  -  -  -  -
op2_contains_op1        -  -  Y  -  -
op2_is_ref_to_op1       -  Y  -  -  -
op2_subclasses_op1      -  -  -  Y  -
aux_match               -  -  -  -  Y  // Invokes auxiliary table, before cross.

[Actions]
do_true                 1  1  1  1  1  // Put op1 before op2.
```

Figure 63: Decision table for evaluating *before* predicate same-side model changes

when it comes to working across replicas. This is because it is often the *second* operation of a pair that will have the last word, so to speak, on the outcome. For example, if two changes rename the same (i.e., matched) attribute, the second change will prevail, making perhaps an *after* predicate more appropriate.

Rather than add a new predicate, we choose to keep the original one, but extend its semantics somewhat. Instead of implying precedence, we interpret *before* to mean that an operation has a *preferred position* with respect to another operation, or that it has a contrary purpose. This interpretation avoids same-side inconsistencies, but also raises cross-side conflicts when appropriate.

The `before_same` sub-table of the divided `before_table` is used by Mirador to test same-side changes for ordering. All pairings of same-side operations, except the mirror pairs, are evaluated against the sub-table's rules. If a pair of operations happen to satisfy a rule, the sole action, `do_true` will return a *true* value indicating that $op1$ claims positional preference over $op2$.

Rule 1 in Figure 63 ensures that $op1$ will be executed before $op2$ and prevent an inconsistency, if the element added by the first change contains the element added by

```
[Table]
before_cross  // Evaluates before(op1, op2) for cross side change operations.
              //
[Conditions]  // Rule#  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
elements_match          Y  Y  Y  Y  N  N  Y  Y  Y  -  -  -  -  -  -  -  -  Y  N  -
op1_is_add              Y  Y  -  Y  -  -  -  Y  N  -  Y  Y  -  -  -  -  Y  -  Y  -
op1_is_delete           -  -  Y  -  Y  -  -  -  N  Y  -  -  Y  Y  -  Y  -  Y  -  -
op1_is_alter            -  -  -  -  -  Y  Y  -  N  -  -  -  -  -  Y  -  -  -  -  -
op1_is_class            -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  Y  -  -  -  -
op1_is_reference        -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  Y  -  Y  -
op1_updates_match       -  -  -  -  -  Y  -  -  -  -  -  -  -  -  -  -  -  -  -  -
op2_is_add              Y  -  Y  -  -  -  -  N  Y  Y  -  Y  -  -  -  Y  -  -  Y  -
op2_is_delete           -  Y  -  -  -  Y  -  N  -  -  Y  -  Y  -  Y  -  Y  Y  -  -
op2_is_alter            -  -  -  Y  Y  -  Y  N  -  -  -  -  -  Y  -  -  -  -  -  -
op2_is_class            -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  Y  -  -  -
op2_is_reference        -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  Y  -  -  Y  -
op2_updates_match       -  -  -  -  Y  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
op1_contains_op2        -  -  -  -  -  -  -  -  -  Y  -  Y  Y  Y  -  -  -  -  -  -
op2_contains_op1        -  -  -  -  -  -  -  -  -  -  Y  -  -  -  Y  -  -  -  -  -
op1_is_ref_to_op2       -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  Y  -  -  -
op2_is_ref_to_op1       -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  Y  -  -  -  -
names_are_same          -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  Y  -
containers_match        Y  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  Y  -

[Actions]
do_true                 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  0
do_false                0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1
```

Figure 64: Decision table for evaluating *before* predicate cross-side model changes

the second. Similarly, rule 2 fires if the class being added by *op*1 is the target of the reference added by *op*2. Rule 3 is effectively the reverse of rule 1: *op*1 is before *op*2 if both targets are being deleted and the first target is contained by the second. Rule 4 was added to handle the subclassing of a newly added class, as done in the running example. Note that there is no need to test `on_same_side` for these rules because it is implied by the nature of the tested conditions—there can be no containment, reference or inheritance across models.

As the right-most rule is the last one tested, it can be used as a default, or catch-all, rule. Thus, all cross-side pairs will fire rule 5, testing condition `aux_match`, which is not a condition at all, but, rather a jump to the auxiliary `before_cross` sub-table that contains the rules for cross-side testing (Figure 64). This *table chaining* makes it possible to modularize the tables, and can be extended to any depth. A wise precaution is to add an error checking subtable to the end of any chain. This way, situations that do not fire an earlier rule will fire one of the rules of the error subtable, or a final rule of all *don't care* conditions. Two actions available for dealing with this eventuality are `do_assert` and `do_throw`.

Although more complex, the cross-side table works in the same manner as the same-side table, except its conditions are applicable to cross model situations. For instance, two matched elements being added to the same container element will fire rule 1, tagging *op1* as being before *op2*. If the reverse is also true, a conflict exists between the two. The `do_false` action of default rule 20 marks a change pair as not being dependent.

Conflict Resolution

Once the change operations have been extracted from the difference models and put into a single list, the models are essentially merged. All that remains is to resolve the key conflicts and order the results. Mirador uses a `resolve_table` to make resolution decisions. The listing of a specification file for this table is given in Figure 65. Automatic conflict resolution of the running example with this table will output the following merge report lines.

```
    --- AUTO RESOLVING CONFLICTS ---
1  [ [sublists:HTMLList {deleted} (WN3I6_223,L)] ]  [3:do_master <]
   [ [sublists:HTMLList {deleted} (WN3I6_223,R)] ]
2  [ [sz {altered} (DJJ3S_961_,L)] ]  [4:do_left < 4:do_right >]
   [ [sz {altered} (DJJ3S_961_,R)] ]
3  [ [sublists:HTMLDocElem {added} (Fu6CN_D25,L)] ]
       [1:do_rename <> 1:do_left < 1:do_right >]
   [ [sublists_rt:HTMLDoc {added} (c3AjT_8,R)] ]
4  [ [HTMLList {altered} (qyd2E_E3_,L)] ]  [6:do_false X]
   [ [HTMLList {altered} (qyd2E_E3_,R)] ]
5  [ [name {deleted} (DJJ3S_861,L)] ]  [3:do_master <]
   [ [name {deleted} (DJJ3S_861,R)] ]
6  [ [name {deleted} (qyd2E_35,L)] ]  [3:do_master <]
   [ [name {deleted} (qyd2E_35,R)] ]
```

The report shows that the delete-delete conflicts of lines 1, 5 and 6 are resolved by rule 3 invoking the `do_master` action, which elects, based on the user-settable "Master side" option of the Element Match Panel, to apply the left side of the conflicts. Line 2 finds, through rule 4 testing the `update_same_property` condition, that the changes being made to attribute `sz` do not overlap, and therefore can both be applied using actions `do_left` and `do_right`. The outcome is similar for line 3: associations added

```
[Table]
resolve       // Executes resolve(conflict_block)

[Conditions]  // Rule#  1  2  3  4  5  6
conflict_is_simple      Y  Y  Y  Y  Y  -
conflicts_match         N  -  -  -  -  -
left_is_add             Y  Y  -  -  -  -
left_is_delete          -  -  Y  -  -  -
left_is_alter           -  -  -  Y  Y  -
right_is_add            Y  Y  -  -  -  -
right_is_delete         -  -  Y  -  -  -
right_is_alter          -  -  -  Y  Y  -
updates_are_equal       -  Y  -  -  Y  -
update_same_property    -  -  -  N  -  -
conflict_with_name      Y  -  -  -  -  -

[Actions]
do_rename               1  0  0  0  0  0  // Rename conflicting objects
do_master               0  1  1  0  1  0  // Execute master side change
do_left                 2  0  0  1  0  0  // Execute left side change
do_right                3  0  0  2  0  0  // Execute right side change
do_false                0  0  0  0  0  1  // Signify conflict is unresolved
```

Figure 65: Decision table for resolving cross-side model change conflicts

to class **HTMLForm** target different elements, but use the same role name. Since the associations were not matched, resolution wants to keep them both, but doing so will result in an invalid model. Rule 1 solves the problem by running `do_rename` to give the elements slightly different names, before executing `do_left` and `do_right`. Line 4 reports that deriving **HTMLList** from different superclasses does not match any rule of table `resolve` other than the catch-all rule 6, which returns *false* to signify that the conflict was not resolved and that a user decision is required.

Customizing Rules, Conditions and Actions

By making decision table definitions accessible, Mirador opens up its conflict handling to modification. However, it can be tricky to get things right. Rule 5 and 6 of table `before_cross` offer a case in point: they work in tandem to properly raise a conflict. If, when evaluating $before(op1, op2)$, it is found that $op1$ deletes an element that $op2$ is attempting to alter, rule 5 will fire, returning *true*. Then, when the evaluation is run in reverse, $before(op2, op1)$, the same situation will cause rule 6 to fire and a conflict will be detected. Without both rules, the conflict would be missed and the element would be deleted, with the work done on the other side being lost.

It is also possible to add a new table condition or action to the Java code for use in any table. This involves either defining and instantiating a **TableCondition** object in which the `testCondition` method has been overridden to perform the desired test, or a **TableAction** object that overrides `doAction` to take the appropriate measures in response to the firing of a rule. The new objects must then be registered with the **DecisionTable** class and provided with labels to identify them to the parser.

The most effective way to extend the tables with custom conditions and actions is to specialize their relevant method using an anonymous inner class, as the code fragments below, for the `on_same_side` condition and `do_true` action, illustrate.

```
/** Decision condition to test for changes on same side of merge. */
public TableCondition on_same_side =
    new TableCondition("on_same_side", new ConditionTest() {
  @Override public boolean testCondition(Object... objs) {
    return ((AtomicChangeOp) objs[0]).getMergeSide()
        == ((AtomicChangeOp) objs[1]).getMergeSide();
  }
});

/** Decision action simply sets return code of result object to true. */
public TableAction do_true = new TableAction("do_true") {
  @Override public boolean doAction(ActionSet steps, Object... objs) {
    steps.setMergeSide(MergeSide.BASE);
    steps.setResult(Tristate.TRUE);
    return true;
  }
};
```

Chapter 8

Validation

Validation of this research concentrates on two distinct aspects: the viability of the proposed approach to model merging, and the ease of performing a reasonable merge. The successful implementation of Mirador provides the primary evidence for the approach's viability. As for ease of use, a series of merge test cases are used to exercise Mirador and another tool. The resulting merge sessions are then treated as case studies for analysis, with the relevant portions being compared and contrasted.

8.1 Hybrid Workflow Viability

The Mirador prototype is evidence of the viability of the hybrid merge workflow. Mirador's components are able to implement the transforming filters of the architecture depicted in Figure 40 and succinctly expressed by the accompanying algebra. The merging tool also manages to integrate the filters into a fully functioning whole. Even cross-tool merges have been successfully executed by the prototype.

One of the strongest features of the workflow is its ability to deal with models from multiple tools. Ironically, this is also its weakest link. The complexity of the Model Normalization and Model Denormalization phases exact a price. While Mirador proves that the necessary code *can* be written, the process is difficult; it must be carried out for both of the phases, for every modeling tool Mirador is expected to handle. The mitigating factor is that it need be done only once per tool.

Evaluation of the merge test cases used to exercise Mirador in the sections that follow also validate the workflow. They demonstrate that the tool is indeed capable of performing reasonable merges. Going deeper, they validate several individual phases of the workflow, both by comparing the likeness of their outcomes to those of a respected MDE tool, and by contrasting differences in the same, justified with arguments from user experience and expectations.

Admittedly, the workflow, its pipe and filter architecture, and the merge algebra are descriptive, rather than prescriptive. They have emerged from a system that, as is the nature of prototypes, evolved from the ground up. Clearly, the design could be better, the separation of phases cleaner, and the application more robust. Still, these valid concerns notwithstanding, nothing has come to light during our experimentation with, and observations of, the hybrid merge workflow to suggest that the approach is anything but viable.

8.2 Primitive Evaluation Cases

This section revisits the primitive evaluation cases used in Chapter 3 to highlight model merging problems with existing MDE tools. Only two tools will be considered here: Mirador and version 8.01 of IBM Rational Software Architect (the latest version at the time of writing). Rather than focus on the merged outcomes, as was done previously, here the emphasize will be on the merge sessions engendered by each tool and the ensuing user experience.

The experiences of a user are important, because bad experiences will in general lead to bad results, and eventually make merging something to be avoided at all costs, diminishing the benefits to be derived from model-driven engineering. User experience is also important, because in many ways, it is the only means of comparing merge tool performance. Objectively comparing the results of tools that rely heavily on user input can be difficult, especially if the tools differ as widely in their construction and behavior as Mirador and RSA.

8.2.1 Methodology

For each evaluation case, the three models involved in the merge will be shown in a screenshot of the RSA model comparison window (e.g., Figure 66). The arrangement is the same as that used in Section 3.2: base model at the top, left and right models below and to either side. The models are shown as trees of model elements, though RSA, which has impressive graphics, can also display them as class diagrams.

Two additional panels appear between the replica-versions; these contain the respective ancestors of the left and right models. Since both replicas came from the same source and no changes have yet been applied to the base model, the models pictured in the ancestor panels and the base panel are identical. The panel at the upper left is a tabbed dialog that displays the conflicts RSA has detected or either of the sets of changes made by the Modelers.

The merge test cases will not be run to completion in RSA, since, with suitable user selections, any of the possible outcomes can be obtained. Instead, the presentation and the choices it offers will be scrutinized. However, with Mirador, which offers an unguided batch mode, the merge *will be* run to completion. The merged result out of Mirador and its merge report will then be examined, and the comparable aspects contrasted with those of RSA in order to draw out distinctions.

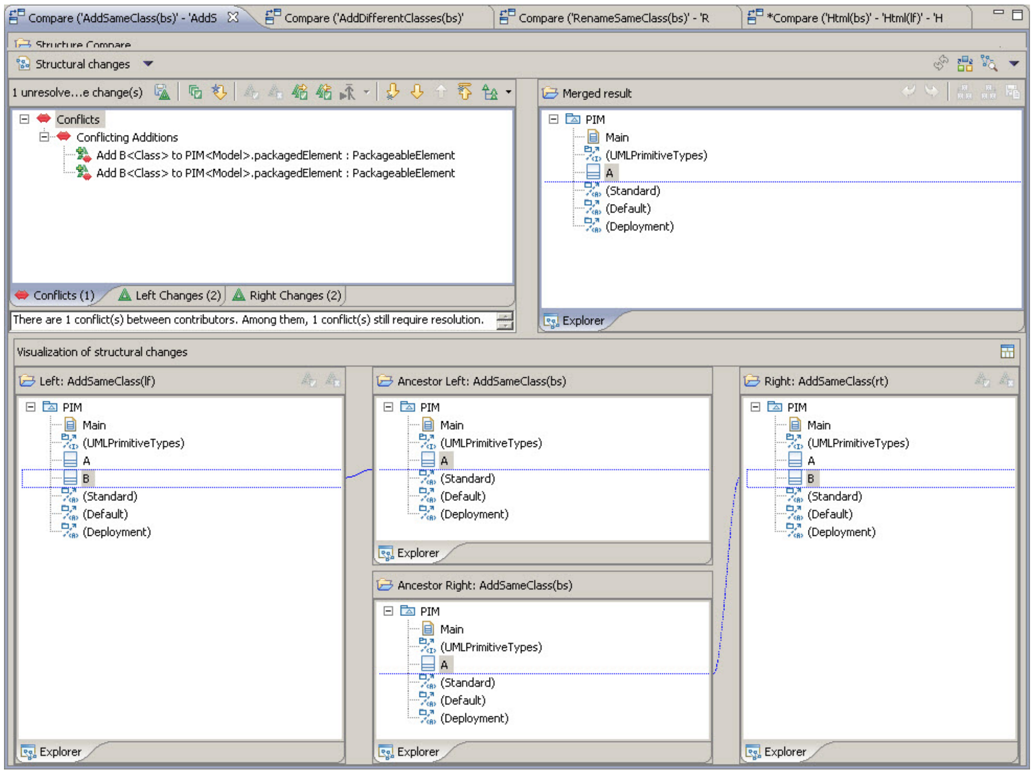
For all the merges that follow, Mirador was set to use the “by ID” matching strategy with a weight of 0.2 and the “by Name” strategy with a weight of 0.7. A matching threshold of 0.6 was selected. The merges were run in batch mode, so there was no opportunity to sway the results through user interaction.

8.2.2 Add Same Class

In the “add same class” case, both Modelers add a class **B** to the base design. The replica-versions are shown in Figure 66, *after* modification in RSA. The merged model, shown *before* application of any changes, is identical to the base model.

Even though the new classes do not share the same ID, RSA matches them as being the same element (an improvement over version 7). This matching is the cause of the contradicting changes seen in the conflict panel. If the Merger does not elect to ignore both changes, they must select one of these alternatives: either add class **B**, or add class **B**(!). Since the tool has decided that the elements are the *same element*, and since, in this instance, they happen to be *identical*, why is the choice of which class **B** to accept offered at all?

Nonetheless, RSA’s conclusion is the correct one, *if* the classes are indeed the


Compare ('AddSameClass(bs)' - 'AddS
Compare ('AddDifferentClasses(bs)'
Compare ('RenameSameClass(bs)' - 'R
*Compare ('Html(bs)' - 'Html(lf)' - 'H
Structure Compare
Structural changes
1 unresolve...e change(s)
Merged result
Conflicts
Conflicting Additions
PIM
Main
(UMLPrimitiveTypes)
A
(Standard)
(Default)
(Deployment)
Conflicts (1)
Left Changes (2)
Right Changes (2)
Explorer
Visualization of structural changes
Left: AddSameClass(lf)
Ancestor Left: AddSameClass(bs)
Right: AddSameClass(rt)
Ancestor Right: AddSameClass(bs)
B

Add B<Class> to PIM<Model>.packagedElement : PackageableElement
Add B<Class> to PIM<Model>.packagedElement : PackageableElement

There are 1 conflict(s) between contributors. Among them, 1 conflict(s) still require resolution.

Figure 66: "Add same class" case being merged in Rational Software Architect

same element. But what if the two classes are not the same element, but just happen to share the same name? Then, not only has RSA imposed a choice on us that is no choice at all, it also prevents us from doing what we desire—to add both classes. Like most tools, there are no provisions in RSA to undo a match, or to apply both changes of what it considers to be a conflict, both of which Mirador allows. We will come across this situation several times in the case studies.

Part of the Mirador merge report for the primitive appears below. The first section reveals that Mirador has detected the same conflict as RSA and for the same reason—identical names. However, because Mirador does a recursive analysis of the element(s) involved in a conflict, it notices what RSA does not—that the elements being added are, in fact, identical. Further, it recognizes that it can resolve the conflict by arbitrarily choosing one of the changes. This it promptly does, taking the change from the "master" side (with left being the default).

For ordering purposes, the conflicting (but since resolved) changes of the primitive are combined in the next section, into a single **ContradictChangeOp** (curly braces) composed of a left and a right **CompositeChangeOp** (square brackets). Resolution leaves only one non-conflicting change of the pair to apply to the base model. This is what the third section of the report shows as occurring. The final section of the report lists the elements of the resulting merged model in a hierarchical fashion—a root package containing one **A** class and one **B** class.

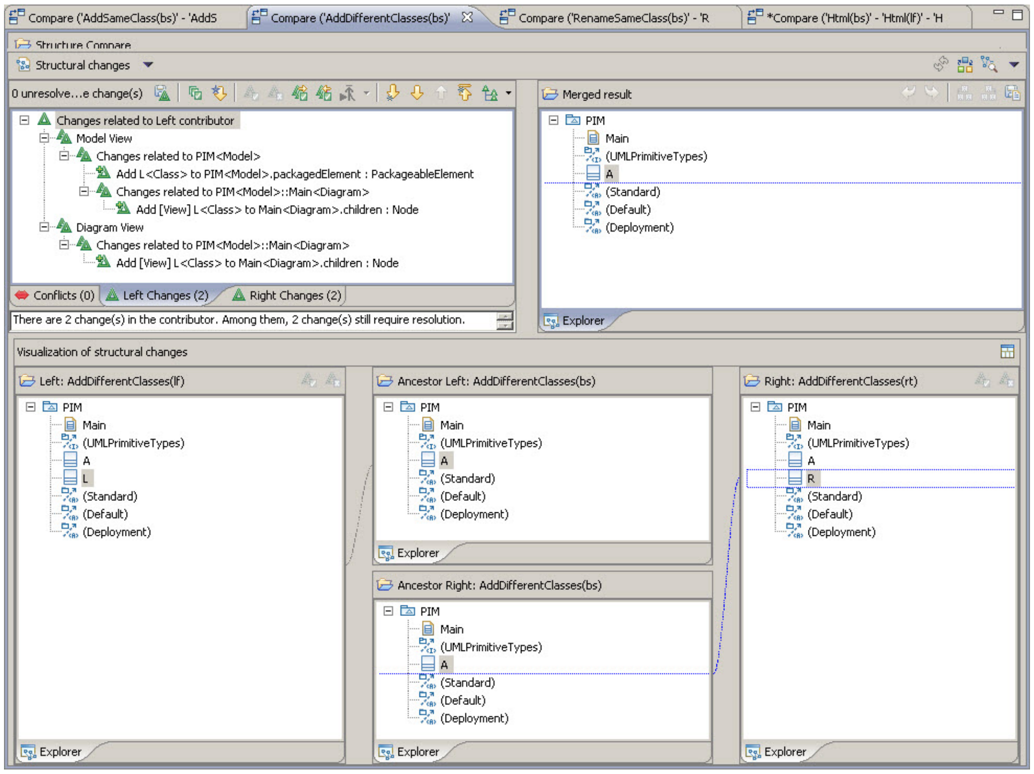
```
        --- AUTO RESOLVING CONFLICTS ---
[ [B {added} (C1CIm_9C,L)] ] [1:do_master <] [ [B {added} (qrJNI_9C,R)] ]
        --- ORDERED & RESOLVED CHANGE OPS ---
{ [ [B {added} (C1CIm_9C,L)] ], [ [B {added} (qrJNI_9C,R)] ] }
        --- APPLIED CHANGE OPS ---
[ [B {added} (C1CIm_9C,L)] ] <<    [ [B {added} (qrJNI_9C,R)] ]
        --- MERGED MODEL ---
add-same-class - ARjm5_92:EPackage
    A - ARjm5_D2:EClass
    B - C1CIm_9C:AddedEClass
```

Mirador's automatic outcome for this evaluation case is perfectly reasonable. The criticism made of RSA with regard to the classes possibly representing different model entities also applies here. The differences with Mirador are that, on the one hand, interactive mode makes it possible to override the matching of the two elements, and, on the other hand, the use of additional matching strategies (e.g., "by Structure" in a model that actually has some structure) might not have matched the elements in the first place. In both these cases, two **B** classes would have been automatically made part of the merged model, which another table action could rename.

8.2.3 Add Different Classes

Primitive "add different class" is the same as the last case, except the names of the added classes are not the same. The screenshot in Figure 67 gives the names of the new classes as **L** and **R**. In this test, RSA rightly does not detect any conflicts. With no conflicts to resolve, the Merger must still make merge decisions (reject or accept) concerning the changes listed in the two change panels. The panel in the figure shows the addition of class **L** on the left; the right panel, adding class **R**, is similar.

The panel contents are a bit busy—one change by the left Modeler has generated three changes for the Merger to deal with. If a trivial change can result in this sort of


Compare ('AddSameClass(bs)' - 'AddS
Compare ('AddDifferentClasses(bs)'
Compare ('RenameSameClass(bs)' - 'R
*Compare ('Html(bs)' - 'Html(lf)' - 'H
Structure Compare
Structural changes
0 unresolve...e change(s)
Merged result
Conflicts (0)
Left Changes (2)
Right Changes (2)
Explorer
Visualization of structural changes
Left: AddDifferentClasses(lf)
Ancestor Left: AddDifferentClasses(bs)
Ancestor Right: AddDifferentClasses(bs)
Right: AddDifferentClasses(rt)

```
Changes related to Left contributor
    Model View
        Changes related to PIM<Model>
            Add L<Class> to PIM<Model>.packagedElement : PackageableElement
            Changes related to PIM<Model>::Main<Diagram>
                Add [View] L<Class> to Main<Diagram>.children : Node
    Diagram View
        Changes related to PIM<Model>::Main<Diagram>
            Add [View] L<Class> to Main<Diagram>.children : Node
```

There are 2 change(s) in the contributor. Among them, 2 change(s) still require resolution.

```
Merged result:                 Left:                  Ancestor Left:         Ancestor Right:        Right:
PIM                            PIM                    PIM                    PIM                    PIM
    Main                           Main                   Main                   Main                   Main
    (UMLPrimitiveTypes)            (UMLPrimitiveTypes)    (UMLPrimitiveTypes)    (UMLPrimitiveTypes)    (UMLPrimitiveTypes)
    A                              A                      A                      A                      A
    (Standard)                     L                      (Standard)             (Standard)             R
    (Default)                      (Standard)             (Default)              (Default)              (Standard)
    (Deployment)                   (Default)              (Deployment)           (Deployment)           (Default)
                                   (Deployment)                                                         (Deployment)
```

Figure 67: "Add different classes" case being merged in Rational Software Architect

clutter, what is the potential for real changes made to a sizeable model? To be fair, two of the changes are actually the same change being shown twice in the tree: once in the "Model view" branch and again in the "Diagram view" branch. So, in reality, this single model change has become not three, but only two changes.

MDE tools must create models (the model) and necessarily, diagrams (the view). While some tools blur the distinction between the two, RSA is not one of them. Despite initial appearances, it goes to some lengths to keep them separate. Breaking each change of the test case into two changes—one to add a class to the model and the other to add it to the view—is an example of maintaining this separation.

With the appropriate use of filters, RSA makes it possible to hide changes that concern only the view, thereby eliminating a great deal of the cognitive merging noise we first complained about in [Barrett et al., 2008]. Filtering, combined with RSA's separation of model and view changes, frees the Merger to concentrate on only the model, without concern for the view. It is not possible to work in the other direction, i.e., nothing exists in the view until it has been added to the model first. This is an entirely sensible restriction.

Mirador, because it only merges models at this time, does not have to worry about the complicating factors brought on by views. Batch execution of the “add different class” primitive yields the following merge report.

```
        --- AUTO RESOLVING CONFLICTS ---
        --- ORDERED & RESOLVED CHANGE OPS ---
R {added} (qrJNI_9C,R)
L {added} (C1CIm_9C,L)
        --- APPLIED CHANGE OPS ---
                          >> R {added} (qrJNI_9C,R)
L {added} (C1CIm_9C,L) <<
        --- MERGED MODEL ---
add-diff-class - ARjm5_92:EPackage
    A - ARjm5_D2:EClass
    R - qrJNI_9C:AddedEClass
    L - C1CIm_9C:AddedEClass
```

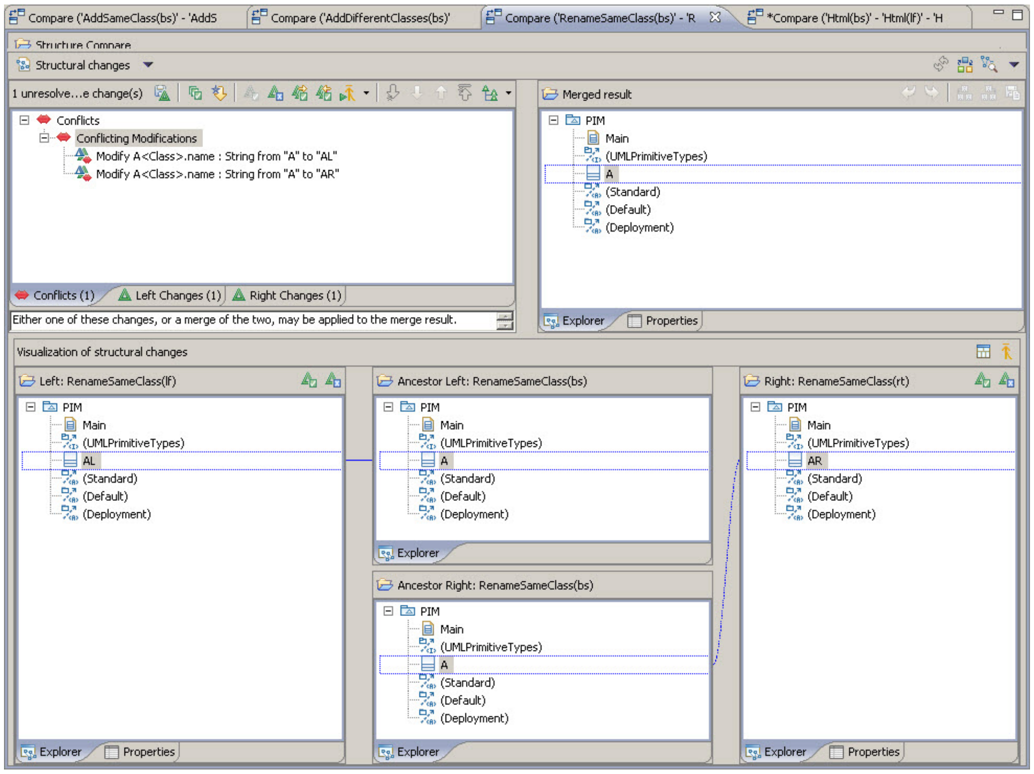
As in RSA, no conflicts are detected. Accordingly, preservation of non-conflicting changes will result in the base model copy being patched with both changes to produce a model containing an **L** class and an **R** class. Again, this is a perfectly reasonable outcome for Mirador’s automatic merging.

8.2.4 Rename Class

The last primitive evaluation case is summarized by Figure 68: class **A** of the base model has been renamed **AL** in the left version and **AR** in the right version. RSA is unable to offer any help with regard to this test case. It can correctly detect that both replica-versions trying to rename the same element is a conflict, but it must hand resolution over to the Merger.

A similar chain of events occurs in Mirador: it detects the same rename conflict, for which no resolution rules exist. Since the only change to be applied to the merged model in this test would be the outcome of the, as yet, unresolved conflict, nothing is applied, and the merged model remains the same as the base model from which it was replicated.

```
        --- AUTO RESOLVING CONFLICTS ---
```


Compare ('AddSameClass(bs)' - 'AddS
Compare ('AddDifferentClasses(bs)'
Compare ('RenameSameClass(bs)' - 'R
*Compare ('Html(bs)' - 'Html(lf)' - 'H
Structure Compare
Structural changes
1 unresolve...e change(s)
Merged result
Conflicts (1)
Left Changes (1)
Right Changes (1)
Explorer
Properties
Visualization of structural changes
Left: RenameSameClass(lf)
Ancestor Left: RenameSameClass(bs)
Ancestor Right: RenameSameClass(bs)
Right: RenameSameClass(rt)
PIM
Main
(UMLPrimitiveTypes)
A
AL
AR
(Standard)
(Default)
(Deployment)

Conflicts
Conflicting Modifications
Modify A<Class>.name : String from "A" to "AL"
Modify A<Class>.name : String from "A" to "AR"

Either one of these changes, or a merge of the two, may be applied to the merge result.

Figure 68: "Rename class" case being merged in Rational Software Architect

```
[ [A {altered} (ARjm5_D2_,L)] ] [5:do_false X] [ [A {altered} (ARjm5_D2_,R)] ]
        --- ORDERED & RESOLVED CHANGE OPS ---
{ [ [A {altered} (ARjm5_D2_,L)] ], [ [A {altered} (ARjm5_D2_,R)] ] }
        --- APPLIED CHANGE OPS ---
[ [A {altered} (ARjm5_D2_,L)] ]   X   [ [A {altered} (ARjm5_D2_,R)] ]
        --- MERGED MODEL ---
add-diff-class - ARjm5_92:EPackage
    A - ARjm5_D2:EClass
```

The only noteworthy difference between the tools with respect to this primitive is that a user of Mirador could conceivably devise a rule that would automatically resolve class naming conflicts (or any other kind of conflict, for that matter). While not likely to be applicable in circumstances as simple as these, it is not too difficult to imagine that changes made to more complicated, domain- or organization-specific models could fall under the coverage of such a specialized rule. For instance, the rule might be: resolve naming conflicts by selecting the change with the name that ends with a camel-case phrase most closely reflecting the name of the element's subsystem.

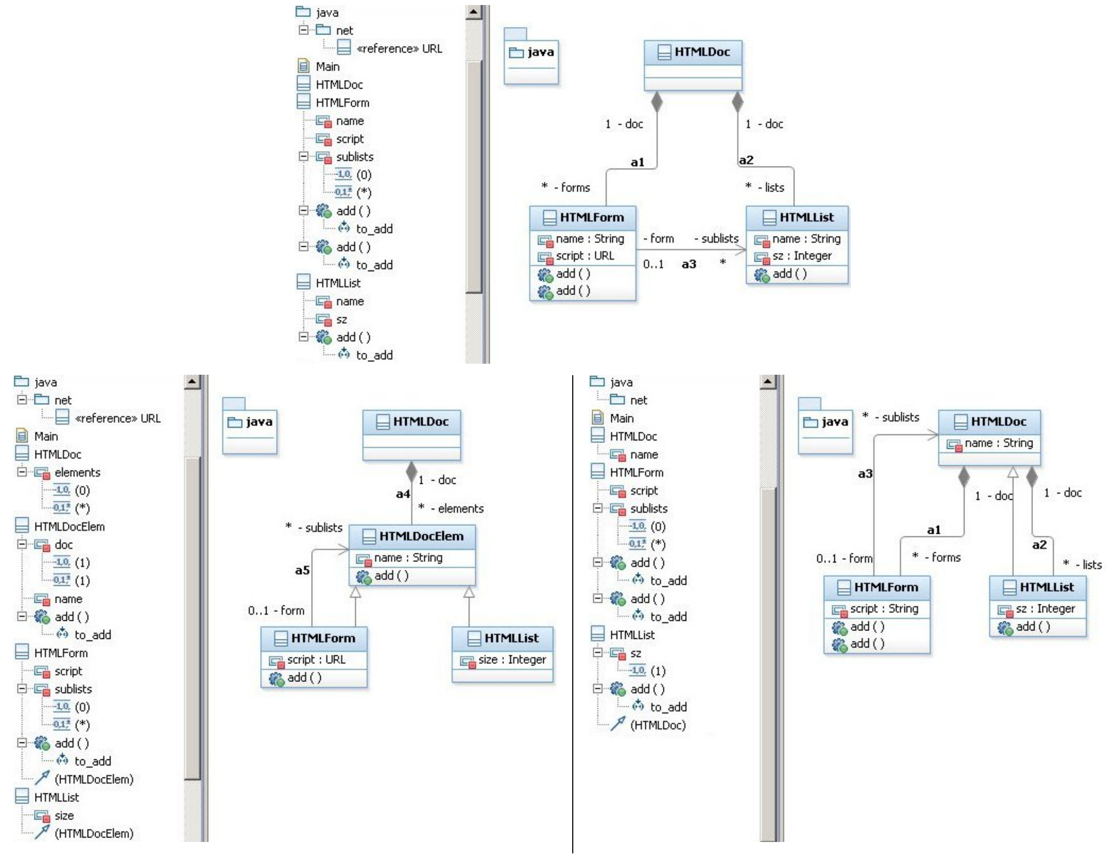


Figure 69: Versions of “HTML Document” model in Rational Software Architect

8.3 Involved Evaluation Case

The model evolution pyramid for the “HTML Document” running example, as rendered by IBM Rational Software Architect, is pictured in Figure 69. The screenshot captures both the element tree and UML views of the model versions. The changes made by each Modeler are the ones enumerated in Section 7.1 and will not be repeated here, except to note two changes not apparent in the class diagrams but revealed by the right tree: the disappearance of class **URL** from package `java.net`, and the establishment of a lower bound of 1 for attribute `sz` (renamed `size` on the left).

Instead of relying on screenshots, as in the previous section, for this evaluation case we construct tables from the logs produced by the two applications over the course of their respective merge sessions. For RSA, diagram changes were filtered out of the log prior to saving. Readers interested in seeing the screens generated by RSA for the case are referred to Appendix D.

Table 3: Left "HTML Document" changes in Rational Software Architect

RL#	Left Changes (16 of 28)
1	Add elements<Property> to HTMLDoc<Class>.ownedAttribute
2	Add <Generalization> to HTMLList<Class>.generalization
3	Add <Generalization> to HTMLForm<Class>.generalization
4	Add sublists<Property> to HTMLForm<Class>.ownedAttribute
5	Add HTMLDocElem<Class> to PIM<Model>.packagedElement
6	Add a5<Association> to PIM<Model>.packagedElement
7	Add a4<Association> to PIM<Model>.packagedElement
8	Delete name<Property> from HTMLList<Class>.ownedAttribute
9	Modify sz<Property>.name : String from "sz" to "size"
10	Delete add<Operation> from HTMLList<Class>.ownedOperation
11	Delete sublists<Property> from HTMLForm<Class>.ownedAttribute
12	Delete name<Property> from HTMLForm<Class>.ownedAttribute
13	Delete add<Operation> from HTMLForm<Class>.ownedOperation
14	Delete a1<Association> from PIM<Model>.packagedElement
15	Delete a2<Association> from PIM<Model>.packagedElement
16	Delete a3<Association> from PIM<Model>.packagedElement

Table 4: Right "HTML Document" changes in Rational Software Architect

RR#	Right Changes (12 of 36)
1	Add name<Property> to HTMLDoc<Class>.ownedAttribute
2	Add <Generalization> to HTMLList<Class>.generalization
3	Add <Literal Integer> to sz<Property>.lowerValue
4	Add [reference] String<Primitive Type> to script<Property>.type
5	Add sublists<Property> to HTMLForm<Class>.ownedAttribute
6	Add a3<Association> to PIM<Model>.packagedElement
7	Delete name<Property> from HTMLList<Class>.ownedAttribute
8	Delete sublists<Property> from HTMLForm<Class>.ownedAttribute
9	Delete name<Property> from HTMLForm<Class>.ownedAttribute
10	Delete [reference] ≪reference≫ URL<Class> from script<Property>.type
11	Delete a3<Association> from PIM<Model>.packagedElement
12	Delete ≪reference≫java.net.URL<Class> from net<Package>.packagedElement

Tables 3 and 4 hold the changes RSA considers to have been made to the left and right models, respectively. The change counts in the headings reflect that the many diagram changes not germane to the scope of this research have been filtered out. The Mirador merge plan—containing all changes *and* conflicts—for case "HTML Document" appears in Table 5. (For fit, "HTML" has been shortened to "H".)

The mapping between the changes of the RSA and Mirador tables is laid out in Table 6. The changes have been grouped by major model element to make them easier to grasp. Changes that conflict are underlined: for Mirador, since the changes are ordered, conflicting pairs have the same number, but not so for RSA, instead, the conflict for a particular change must be read from Table 7 (e.g., according to entry

Table 5: "HTML Document" merge plan of changes and conflicts in Mirador

M#	Left (21 of 21)		Right (9 of 9)
1	to_add {deleted} (qyd2E_A4)	<	
2	HDocElem {added} (sCE3W_4B4)	<	
3	add {added} (sCE3W_0C4)	<	
4	to_add {added} (sCE3W_1C4)	<	
5	lists:HList {deleted} (YXqW0_6F1)	<	
6	doc:HDoc {deleted} (YXqW0_5F1)	<	
7		>	script {altered} (DJJ3S_661_)
8	name {added} (sCE3W_DB4)	<	
9		>	URL {deleted} (McEU4_76)
10	forms:HForm {deleted} (WnHJn_291)	<	
11	elements:HDocElem {added} (Fu6CN_B15)	<	
12	doc:HDoc {added} (Fu6CN_A15)	<	
13	to_add {deleted} (qyd2E_15)	<	
14	doc:HDoc {deleted} (WnHJn_191)	<	
15	add {deleted} (qyd2E_94)	<	
16	HForm {altered} (qyd2E_63_)	<	
17	add {deleted} (qyd2E_05)	<	
18		>	name {added} (c3AjT_11)
19	name {deleted} (qyd2E_35)	<	name {deleted} (qyd2E_35)
20	sublists:HDocElem {added} (Fu6CN_D25)	<>	sublists:HDoc {added} (c3AjT_8)
21	sz {altered} (DJJ3S_961_)	<>	sz {altered} (DJJ3S_961_)
22	name {deleted} (DJJ3S_861)	<	name {deleted} (DJJ3S_861)
23	sublists:HList {deleted} (WN3I6_23)	<	sublists:HList {deleted} (WN3I6_23)
24	HList {altered} (qyd2E_E3_)	×	HList {altered} (qyd2E_E3_)

RC24, left RSA change RL12 conflicts with right change RR9).

Many of the mappings are obvious, for instance, the addition on the left of class **HTMLDocElem** by change #5 in RSA (RL5) maps to Mirador change #2 (M2), which does the same element addition. As Mirador's merge report is somewhat sparing of detail, other mappings can require some investigation to work out, e.g., the deletion of attribute **HTMLList**.`name` in RSA by change RR7 maps—by entity ID—to Mirador M22, and not M19, which deletes **HTMLForm**.`name`.

Mapping a few types of changes, especially those regarding associations, requires taking metamodel clashes into account. Out of the box, RSA, which is based on Eclipse, conforms to Eclipse's UML2 metamodel. A UML2 association consists of an **Association** metaobject, along with a property in each of the linked metaobjects to hold the role name of its end of the association. Mirador uses the Ecore metamodel, which constructs an association out of one or two—depending on whether uni- or bidirectional—**EReference** metaobjects. Because of this, RSA requires only one change (RL14) to remove association `a1` from the base model. Mirador, however,

Table 6: “HTML Document” change mapping between RSA and Mirador

Mirador		RSA		Model Change Action
M#		RL#	RR#	
	18		1	Add attribute `name` to **HTMLDoc**
2		5		Add class **HTMLDocElem**
3		5		Add operation `add` to **HTMLDocElem**
4		5		Add parameter `to_add` to operation `add` in **HTMLDocElem**
8		5		Add attribute `name` to **HTMLDocElem**
1		13		Delete parameter `to_add` from operation `add` in **HTMLForm**
15		13		Delete operation `add` from **HTMLForm**
16		3		Generalize **HTMLForm** from **HTMLDocElem**
19		12		Delete attribute `name` from **HTMLForm**
	19		9	Delete attribute `name` from **HTMLForm**
21		9		Change attribute `sz` to `size` in **HTMLForm**
	21		3	Change lower bound of attribute `sz` to 1 in **HTMLForm**
	7		10,4	Change attribute `script` type to **String** in **HTMLList**
13		10		Delete parameter `to_add` from operation `add` in **HTMLList**
17		10		Delete operation `add` in **HTMLList**
22		8		Delete attribute `name` from **HTMLList**
	22		7	Delete attribute `name` from **HTMLList**
24		2		Generalize **HTMLList** from **HTMLDocElem**
	24		2	Generalize **HTMLList** from **HTMLDoc**
	9		12	Delete class **URL**
5,6		15		Delete association `a2`, roles `doc` and `lists`
12,11		1,7		Add association `a4`, roles `doc` and `elements`
14,10		14		Delete association `a1`, roles `doc` and `forms`
20		4,6		Add association `a5`, roles `form` and `sublists` (unidir.)
	20		5,6	Add association `a3`, roles `form` and `sublists` (unidir.)
23		11,16		Delete association `a3`, roles `form` and `sublists` (unidir.)
	23		8,11	Delete association `a3`, roles `form` and `sublists` (unidir.)

takes two separate changes (M10 and M14) to accomplish the same thing.

Some Mirador changes map to the same RSA change, e.g., RSA bundles all of the changes made to **HTMLDocElem** (M2, M3, M4 and M8) into change RL5, which adds the class to the model. The one added (M4) and two deleted operation parameters (M1 and M13) are treated by RSA in the same way. This bundling of contained model entities reduces the number of changes, but with some loss of change granularity. This does not matter much for deletion (the parameters need to go if their operation is being deleted), but for addition it might. For instance, it would not be possible to accept an added class from one side while accepting an attribute added to its matched class on the other side.

As for conflicts, Mirador detects the six shown in the last rows of its merge plan. The resolution decision table recognizes that the three matched, delete-delete changes

Table 7: “HTML Document” conflicts in Rational Software Architect

RC#	Conflicts (27)
1-11	Conflicting Modification and Deletion
12	Conflicting Modification
13-15	Conflicting Modification and Deletion
16	Conflicting Deletes
	Delete [View] (HTMLDoc<Class>)(HTMLList<Class>)a3<Association>
	Delete [View] (HTMLDoc<Class>)(HTMLList<Class>)a3<Association>
17	Conflicting Modification and Deletion
	Delete [View] (HTMLDoc<Class>)(HTMLList<Class>)a3<Association>
	Modify <Location>.y : EInt from -869 to -974
18-22	Conflicting Modification and Deletion
23	Conflicting Deletes
RL8	Delete name<Property> from HTMLList<Class>.ownedAttribute
RR7	Delete name<Property> from HTMLList<Class>.ownedAttribute
24	Conflicting Deletes
RL12	Delete name<Property> from HTMLForm<Class>.ownedAttribute
RR9	Delete name<Property> from HTMLForm<Class>.ownedAttribute
25	Conflicting Deletes
RL11	Delete sublists<Property> from HTMLForm<Class>.ownedAttribute
RR8	Delete sublists<Property> from HTMLForm<Class>.ownedAttribute
26	Conflicting Additions
RL4	Add sublists<Property> from HTMLForm<Class>.ownedAttribute
RR5	Add sublists<Property> from HTMLForm<Class>.ownedAttribute
27	Conflicting Deletes
RL16	Delete a3<Association> from PIM<Model>.packagedElement
RR11	Delete a3<Association> from PIM<Model>.packagedElement

(M19, M22 and M23) are not really in contradiction (i.e., the desired end is achieved if one, and only one, of the deletes is executed) and resolves them accordingly. Even though RSA finds the same three conflicts (lines RC23-25 in Table 7), it does not attempt to resolve them. Related to RSA conflict RC25, is RC27—a byproduct of UML2 representing associations with an **Association** metaobject. Mirador sees no conflict here, because Ecore does not represent associations per se, just their ends.

There is no RSA equivalent to conflict M21. Mirador detects that attribute `sz` is being altered on both sides, but in a non-overlapping way. Therefore, it recommends that both changes be applied. In RSA, changes RL9 and RR3 affect different *properties* of the same element, and so are not considered to be in conflict to begin with. The outcome is the same in both tools, but in Mirador, because the changes show as an auto-resolved conflict, the fact that the element has been modified by both Modelers is made explicit, alerting the Merger without demanding a decision.

Mirador and RSA both see the addition of two `sublists` references to class

HTMLForm as being in conflict (M20 and RC26), but for different reasons. In RSA, the elements are matched by name, and thus considered to be the same association, so only one is allowed. Had Mirador matched the elements, it would have followed the same course, but due to structural differences (the associations reference different classes) it did not. However, recognizing that a class with two references of the same name is a problem for code generation, Mirador still raises a conflict. The table rules resolve the conflict automatically by giving each reference a slightly modified role name—an appended "_lf" or "_rt", respectively—before applying both changes. This preserves both Modelers' changes, while keeping the model valid.

Conflict M24 is one Mirador objects to, but RSA does not. The left change generalizes class **HTMLList** from **HTMLDocElem**, while the right change generalizes it from **HTMLDoc**. While not strictly a modeling error, this multiple inheritance is another problem for code generation, if the target language is Java, as it is in RSA. With this understanding, Mirador flags a conflict. Unfortunately, as no decision table rule matches the pattern of the conflict, it is left to the Merger to resolve.

There are 22 more RSA conflicts shown in Table 7. All have to do, directly or indirectly, with changes made to the view of the model. There is no way to filter diagram changes out of the conflict list. Most are shown as collapsed, but two have been opened to give a feel for their nature. RC16 is the view counterpart of RC27, which deletes association `a3`. RC17 is typical of the majority of RSA view conflicts: a matched element is deleted from the model on one side and simultaneously altered in the view on the other, both of which cannot happen. Such concerns are is not within Mirador's scope.

For this test case, Mirador was able to detect all of the model conflicts that RSA did, plus one that RSA did not. In addition, Mirador was able to successfully resolved five out of the six conflicts with no user input whatsoever. With a larger and more polished rule base, Mirador can be expected to further leverage semantics for better conflict detection, and more automatic conflict resolution.

Chapter 9

Related Work

Synchronizers and mergers have been either state-based, relying on a static view of data items, or operation-based, depending on dynamic traces of changes. We have, of course, attempted to blend the best of both these approaches.

Static, state-based synchronizers are exemplified by the likes of rsync [Tridgell and Mackerras, 2011] and its replacement, Unison [Pierce and Vouillon, 2004], whereas the more ambitious IceCube [Kermarrec et al., 2001] is an example of a dynamic, operation-based *platform* for data reconciliation. A method for model synchronization based on transformations has been proposed by Xiong et al. [2007]. Study of the behavior of these synchronizers has enabled us to give a definition of synchronization that distinguishes it from merging.

Our definitions of synchronization and merge operations are from the point of view of the items being reconciled, or aligned, and the outcome of the operation. In an attempt to provide more solid footing, Brunet et al. [2006] treat a merge as an algebraic operator over model and model relationship data types. They go on to define a number of other useful model management operators, most of which we have incorporated, along with our own additions, into definitions of the phases of our proposed hybrid merge workflow. Boronat et al. [2007] have gone further, and provided general semantics for the *merge* operator as it applies specifically to class diagram integration, which our work does not take advantage of.

A case for state-based merging is presented by Westfechtel [2010], who argues that operation-based approaches suffer from *missing comparisons* in model differencing and *missing state information* in change weaving (i.e., the actual act of merging).

Thanks to its hybrid workflow, Mirador manages to avoid both deficiencies, the first because it has full access to the states of the replica-versions when differencing, and the second because the operations it manipulates do not come from a change log, but are extracted from model state. Westfechtel's formally defined, context-free merge rules for EMF models (i.e., Ecore) were published too late for us to translate into Mirador decision table rules.

Vendor modeling tools with merge capabilities are predominately state-based and are typified by the three in our survey: IBM Rational Rose, IBM Rational Software Architect, and Enterprise Architect. Others include MagicDraw UML and Borland Together. Several of these tools make use of the state-based Eclipse EMF Compare facility [EMF Compare, 2010]. One operation-based modeling tool is the Fujaba research effort [Fujaba, 2009], which uses its text-based CoObRA versioning software to effect merges [Schneider et al., 2004]. None of the merging done by these tools, or any others to our knowledge, blend the main merge approaches as we have done in Mirador.

Though mostly ignored by vendors, operation-based merging was advanced by Lippe and van Oosterom [1992] as a means for managing changes to object-oriented databases. They describe a merge as a weave of primitive, or atomic, subtransformations taking place in a transformation grid having single- and multiple-valued node types; relate non-commuting operations to conflicts, tabulated in a conflict matrix; and propose three merge algorithms that hinge on the use of a *before* predicate for conflict detection. They also explain a technique for ordering changes and conflicts by partitioning them into blocks. Work done in the area of graph transformations (e.g., [Ehrig et al., 2006]), specifically with regard to critical pairs, and term and graph rewriting, touches on similar concepts.

In Mirador, we have reoriented Lippe and van Oosterom's transformation grid to become our change plane and added another dimension to accommodate matching strategies, thus forming our merge space. We clarify the semantics of their *before* predicate used to populate the plane, so as not to miss certain types of conflicts. We also make a distinction between direct and indirect multiple-valued nodes, with the former identifying what we term key conflicts. Their conflict matrix has been extended by us to occupy four quadrants in order to consider both same-model and cross-model changes. Finally, we have added the technique of conflict collapse to their partitioning scheme in order to break cycles that can occur in the change graph.

Renewed interest in the operation-based approach can be seen in the detection of conflicts as proposed by Koegel et al. [2009], who use operation commutativity, which they term *serialization*, to detect conflicts as we do, but who, for performance reasons, calculate only an approximation, rather than test all change pairings. The *transaction-based merging* of Schmidt et al. [2009] treats sequences of changes combined into transactions as full-fledged data objects subject to version control.

Properly matching model elements is a problem of establishing identity and is vital to successful merging. Working on databases, Lippe and van Oosterom did not concern themselves with matching; others (e.g., [Pottinger and Bernstein, 2003]) take it as a given. While many tools insist on unique element identifiers [Barrett et al., 2008], more robust and flexible ideas are offered by Xing and Stroulia [2005] in the form of the UMLDiff algorithm, which matches based on name and structural similarity, and Kolovos [2009], who furnishes a general comparison language for which Mirador has a specialized evaluator.

The Kompose framework consists of a metamodel specific matching portion, and a generic merging potion [Fleurey et al., 2007]. The matching, which relies on specialized signatures of element type and equality tests, has the potential to use different matching strategies, but only one per type, whereas Mirador supports up to seven. Kompose makes the common assumption that its way of matching is the *only* way and, as with many tools, cannot be overridden. The merging is state-based and has little overlap with our work. Also, its conflicts must be manually resolved, where Mirador can recognize and resolve many conflicts automatically.

SiDiff, as described by Schmidt and Gloetzner [2008], is a framework built around an algorithm by Kelter et al. [2005] for building differencing tools. It does similarity-based model element comparisons before taking the model difference. A kernel configuration must be created from available algorithms in order to specify how to compute the similarity for each type of diagram element. The algorithms of SiDiff are comparable to Mirador's matching strategies, but like Kompose, can only be applied one strategy per diagram type.

SiDiff has been used by Treude et al. [2007] for differencing large models. They show how to reduce the inherent $O(n^2)$ complexity of model element comparison to $O(n \log n)$. Unfortunately, SiDiff appears to be a dead project. Several attempts to contact the team went unanswered, and the one answer was not followed-up by them. The difference metamodel of Cicchetti et al. [2007] that was extended by us for use with Mirador has been tied into another metamodel for conflict specification in [Cicchetti et al., 2008], which we do not yet make use of.

Lastly, a useful resource for discovering new conflict patterns, which can then be encoded as rules for Mirador's decision tables, has recently appeared on the Web. The project called Colex, a contraction of "conflict lexicon" [Brosch et al., 2010b], aims to be a single point of reference for conflicting merge scenarios. It is comparable to our catalog of primitive evaluation cases but, being collaborative in nature, has the potential for greater growth.

Chapter 10

Conclusion

This work started off with an investigation into the nature of model merging and the model merging session. Exploration of how merging has been addressed in several state-of-the-art MDE tools provided a baseline of understanding, as well as a feeling for how the merge session progresses and for some of the attendant problems. Along the way, a set of primitive evaluation merge cases was used to expose weaknesses of the tools and the issues underlying how they address the merge process.

To lend an air of familiarity to the process, the primitives were also merged using standard text-based tools. It was from this familiar viewpoint, and its accompanying expectations of what constitutes normal merge behavior, that we argued that much of what occurs when models are the operands of a merge operation, violates developer expectations and, indeed, is often counterintuitive.

To test the proposition put forward by the thesis statement of this research, an alternative workflow to those imposed by the main approaches to model merging was described and its phases defined with algebraic operators. The resulting merge workflow is a hybrid that employs the best merging approach for the task at hand.

As an aid for visualizing and reasoning about aspects of the hybrid merge workflow, a coordinate system for a three-dimensional merge space was described. Two of its axes are the change operations of the composite model transformations effectively executed by developers on the left and right replicas of a shared base model, respectively. The coordinate of the space's third dimension is the various matching strategies used to map the model elements of these new versions to one another.

The hybrid workflow along with concepts from the merge space have found realization in Mirador, our model merging tool. An overview of Mirador's architecture was given, highlights of its design elaborated on, and some features of its implementation discussed. A description of how the implementation was validated against the merge capabilities of the latest version of a top-tier MDE modeling tool was also provided.

10.1 Summary of Contributions

The evidence that this research makes the contributions claimed in the introduction is consolidated in this section. It details how the novel aspects of Mirador and its unique hybrid merge workflow address the issues of the problem statement, and to what degree they meet the stated research objectives. For traceability, footnote callouts in the text tie statements made here back to list items in the introduction.

The *definition of the hybrid merge workflow* is the conceptualization of the thesis statement, and the *implementation of the prototype* Mirador, its materialization. Together, the two form the major contributions of this research, with the second validating the first. The normalization workflow phase makes the core functionality of Mirador and its manipulation of models *independent of modeling tools and their metamodels*,[1] and *decouples it from any recorders of change operations*.[2] Defining the workflow phases with algebraic operators directly supports a research objective,[a] as does change recorder decoupling.[d]

Establishing an accurate correspondence between elements of the participating models is critical to the success of a merge. Rather than settle on one kind of mapping, as most, if not all, merging tools do, Mirador *uses multiple matching strategies*,[3] including user-defined ones. This synthesis of model comparison techniques allows for the characteristics of the models themselves to determine the best measure of similarity for the elements under consideration.[c] Which strategies are available for matching is under user control, and any merge decisions made by Mirador may be overridden.[c] Lastly, matching done in a prior merge session may be saved for use in guiding the comparison of the same models in future merge sessions.

Asking the user to step in whenever a difficulty, no matter how slight, is encountered is not only irksome, but can inundate the user with irrelevant details. Furthermore, it makes getting early merge feedback laborious. Mirador *minimizes, or even* *eliminates, the need for human input.*[5] This enhancement of tool usability is achieved by removing from consideration as many changes and conflicts as possible without the loss of merge choices,[b] and by making as much of the merged context available as possible before forcing a choice, thus aiding more informed merge decision-making.

A workflow that obtains change operations from model state, rather than from a log, removes some changes from consideration, and thus keeps the number of changes to a minimum, consequently lowering the probability of conflict. The auto-resolution of conflicts that should never be brought to the user's attention (e.g., deletes of the same element) also improves usability. More context is made available for merge decisions through the ordering of changes. This ensures that dependent changes do not come up for consideration until their dependencies have been executed. The ordering is achieved with change partitioning and, in the case of cycles, our technique of conflict collapse.[8]

The workflow as devised, not only potentially reduces the cognitive load on the user, it can also make do with no user at all. This non-interactive "fast path" through the workflow provides for Mirador's batch mode of operation.[b] The execution path automatically matches model elements, resolves conflicts, and orders changes so as to apply as many patches to the base model as possible; generating a merged model and full merge report in the process. It halts after all changes have been applied, or upon encountering an unresolved conflict or error. The mode is ideal for obtaining rapid merge feedback, and is suitable for script or batch processing.

Mirador successfully demonstrates the feasibility and effectiveness of using *decision tables to drive conflict detection and resolution.*[4] Its decision table mechanism allows for the coding of table rules to recognize and handle common merge situations, thereby improving merge results.[g] Witness Mirador's execution of the "add same class" evaluation case: there were no duplicated elements in the merged model nor any conflicts to resolve. To date, we know of only one other tool that has been able to equal this ideal.

The knowledge of common situations and troublesome merge patterns that resides in Mirador's decision tables is readily extended by the addition of more rules,[h] as demonstrated by the "HTML Document" example, in which a rule was added to detect multiple inheritance conflicts. Rules may also be added to deal with a specific class of models, or in response to local or domain-specific needs.[e] The mechanism also allows for expanding the conditions and actions of the tables, bringing the full power of Java to bear on improving merge results. For instance, the deep compare of elements done by the `update_same_property` condition can determine if alters of the same element overlap or not. If not, the existence of an `add_both` action lets both changes occur, something no other tool encountered allows.

The extensibility offered by Mirador's decision table mechanism *opens the door to semantic merging* as understood by Mens [2002] and Brosch [2009].[g] The tables exploit semantics to a small degree in the detection of conflicts, in which the meaning of an update is used to filter out syntactic conflicts that are not actually in conflict. Rule-based merging with its potential for using semantics, along with the decoupling front-end of the workflow, and the normalization of its internal models, makes the core of Mirador a powerful, *general-purpose model merging engine.*

In addition to the above mentioned contributions, the research makes some minor ones. Though concerned primarily with merging, we offered complementary definitions of the horizontal types of synchronization and merging in Section 2.1, because such appear to be lacking in the literature.[6] The primitive evaluation cases used to exercise the vendor tools introduced in Chapter 3 are a few of a handful of similar cases we have defined. Though hardly the complete catalog we had set out to create,[9] the cases do document several difficult merge situations.[f]

Finally, several devices, originally conceptual in nature, either proposed by us (change plane, merge space), or by others and extended by us (transformation grid, conflict matrix, and conflict partition), have found a home within the code of Mirador. Combining the two dimensions of the change plane—a graphical means for expressing a merge of change operations[a]—with a third for matching strategies creates a space within which to reason about a merge.[7] The two-dimensional technique of conflict detection using multiple-valued nodes was extended into this third dimension in order to detect matching strategy conflicts.

10.2 Limitations of Work

There are two sides to our work: the theoretical and the practical. Both have limitations, some of which have already been mentioned in passing, but which are discussed here in more detail. Reasons for these limitations have to do with restrictions in the scope of the research, certain design decisions, and incomplete features of the prototype implementation.

Discussions of merging and merge tools tacitly assume that there is no loss of generality in considering only two replicas at a time; our work is no different in this regard. While it is true that repeated application of the merge operation can always keep the number of replicas involved to two, it is not true that the order of application is of no importance. This is because *merging is not associative.* To our knowledge, no one has conducted an analysis of this fact. Experience leads us to believe that the ramifications for merging in practice are small: Any merge that needs to be done in response to a commit can usually be completed before the next commit, and, if not, then the merges may simply be serialized.

The thesis does not try to define the model produced by the hybrid workflow, nor what model properties the process should preserve. The transition from operation-based to state-based models that occurs in the normalization phase of the hybrid workflow involves a loss of information. For this reason, the operation extraction phase is unable to reconstruct, exactly, the operational trace of changes that were executed against a replica. This loss of intermediate history is more than offset by a reduction in bookkeeping and merge complexity. However, it does mean that any matching strategies based on actual change history, as first proposed by us [Barrett et al., 2010a], will not be available.

Though Mirador is adequate as a proof of concept, in its current form, it has shortcomings that make it unsatisfactory as a general-purpose model merging tool, and limited our discussion of some more general issues. Most seriously, the use of Ecore for the normal-form representation of its internal models restricts it to merging only class diagrams. Also, Ecore cannot model some common UML concepts, such as element visibility. At this time, the only input decouplers that have been built are for Ecore and Fujaba models, and on the output side, only Ecore is supported.

Element matching strategies and decision tables are two areas where there was simply not enough time to do them justice. Only three strategies have been implemented in full and one of those, “by ECL”, is not intrinsically a strategy but, rather, a means of loading a *script*, that is the strategy. When matching, no attempt is made to use the best strategy for the elements under consideration. Instead, the same strategy is used for all elements: either the strategy measure chosen by the Merger, or the combined overall measure. Discoveries of missed conflicts (since corrected) have shown that the decision table rules are incomplete. Subtle interactions between the rules can make it difficult to modify the tables without introducing unwanted (or removing wanted!) side effects, or cause inconsistencies. No means of systematically testing the tables has been devised, though assertions and exceptions may be added in the form of chained, error checking subtables.

The graphical user interface is not fully developed, particularly in terms of its display of the models, and the change operations and conflicts. During change application, applying changes that follow a rejected change may result in an inconsistency. Mirador flags the error only after the fact, doing no checking prior to patching the base model that could prevent it. Finally, the merge session report is rudimentary at best, being essentially a dump of Mirador debug messages, and requires too much tool and model specifics in order to interpret properly.

10.3 Suggestions for Future Research

Aside from those subject to an immediate fix, the limitations and shortcomings noted in the last section call attention to several areas that we deem worthy of further improvement and investigation. The suggested topics have been placed under the following six broad headings.

Merge Engine Architecture As a proof of concept, Mirador exhibits a bottom-up architecture. A new architecture could provide the merge engine with well-defined interfaces. This would allow the engine to be dropped between any pair of model decoupler/coupler modules that implement the proper interfaces, giving the engine the ability to then merge their models. The coding of such module pairs to support some of the more popular tools would be a natural part of this improvement.

Moving from Ecore to UML2, the EMF-based metamodel of the Unified Modeling Language [UML2, 2010], for the normal-form representation of models would allow UML concepts to be fully supported, and enable the merging of more diagram types. Further, the leveraging of existing mathematical mappings of UML in order to prove merge correctness is an intriguing possibility.

Model Element Matching The comparison strategies Mirador uses to match model elements are a promising area of focus. Completing implementation of the placeholder strategies that the tool now uses is a necessary first step. Afterwards, forming a meaningful interpretation of the similarity measures, their weights, and how best to use the strategies in combination could be investigated. The matching mechanism itself needs to be updated to select the best matching strategy for any pair of elements under consideration. Another decision table could be constructed to drive this selection.

Change Conflict Modeling The conflict detection and resolution phases of the hybrid workflow only recognize one type of conflict, namely, contradicting change operations (i.e., those detected by the *before* predicate). Other types that are part of a conflict model proposed by Brosch et al. [2010a] are operation contract violations and post-merge violations. The abstract **ConflictChangeOp** was put into Mirador's change operation hierarchy (Figure 61) as an extension point for supporting this model. The metamodel we use for differencing has been extended by Cicchetti et al. [2008] to support conflict modeling as well and is another avenue to consider. Finally, conflict visualization for Mirador is the subject of an ongoing informal collaboration.

Decision Tables Experience has shown that modifying the decision table rulebase by hand can be difficult. If the flexibility of rule customization is to be more generally usable, a tool for table rule maintenance must be devised. The same can be said for table conditions and actions, which at present can only be created or altered by editing the Java code. Rule-based technologies are an alternative to using tables for decision-making. The Epsilon Comparison and Merging Languages [Kolovos et al., 2010], which are essentially rule engines for models, are strong candidates for filling this role.

Semantic Merging The path to full-fledged semantic merging is not clear at this time. Nor is it clear whether decision tables alone are enough to achieve it; they will likely need to work in conjunction with other mechanisms. For example, in [Barrett et al., 2009], we discuss the merging of use case models that have been given a formal syntactic structure. Formal semantics are imparted to the models through finite state machines, for which an equivalence operator is defined for the detection of merge conflicts. A similar approach applied to the merging of other model types might enable a significant increase in the role of semantics in model merging.

Mirador Merging Tool The prototype tool that has emerged from this research has shown the viability of the hybrid merge workflow, and demonstrated numerous ideas for the improvement of model merging, both in terms of results and in terms of usability. Any of the above suggested research could benefit Mirador greatly, and the possibilities are exciting. But most exciting of all is the possibility that its contributions to model merging could play a small role in helping model-driven development achieve its potential.

Bibliography

Stephen Barrett, Patrice Chalin, and Greg Butler. Model merging falls short of software engineering needs. In *MoDSE '08: Internat. Workshop on Model-Driven Software Evolution*, Apr 2008.

Stephen Barrett, Daniel Sinnig, Patrice Chalin, and Greg Butler. Merging of use case models: Semantic foundations. In *TASE '09: Internat. Sympos. on Theoretical Aspects of Software Engineering*, pages 182–189, Jul 2009.

Stephen Barrett, Greg Butler, and Patrice Chalin. Mirador: a synthesis of model matching strategies. In *IWMCP '10: Internat. Workshop on Model Comparison in Practice*, pages 2–10, Jul 2010a.

Stephen Barrett, Patrice Chalin, and Greg Butler. Decoupling operation-based merging from model change recording. In *ME '10: Internat. Workshop on Models and Evolution*, pages 23–32, Oct 2010b.

Christian Bartelt. Consistence preserving model merge in collaborative development processes. In *CVSM '08: Internat. Workshop on Comparison and Versioning of Software Models*, pages 13–18, New York, NY, USA, May 2008. ACM. ISBN 978-1-60558-045-6. doi: http://doi.acm.org/10.1145/1370152.1370157.

Lars Bendix, Maximilian Koegel, and Antonio Martin. The case for batch merge of models – issues and challenges. In *ME '10: Internat. Workshop on Models and Evolution*, pages 102–113, Oct 2010.

Artur Boronat, José Á. Carsí, Isidro Ramos, and Patricio Letelier. Formal model merging applied to class diagram integration. *Electron. Notes in Theor. Comput. Sci.*, 166:5–26, Jan 2007. ISSN 1571-0661. doi: 10.1016/j.entcs.2006.06.013.

Petra Brosch. Improving conflict resolution in model versioning systems. In *ICSE '09: 31st Internat. Conf. on Software Engineering, Companion Volume*, pages 355–358, Washington, DC, USA, May 2009. IEEE Computer Society. ISBN 978-1-4244-3495-4. doi: 10.1109/ICSE-COMPANION.2009.5071020.

Petra Brosch, Horst Kargl, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Gerti Kappel. Representation and visualization of merge conflicts with UML profiles. In *ME '10: Internat. Workshop on Models and Evolution*, pages 53–62, Oct 2010a.

Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. Colex: a web-based collaborative conflict lexicon. In *IWMCP '10: Internat. Workshop on Model Comparison in Practice*, pages 42–49, Jul 2010b.

Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, and Mehrdad 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. ISBN 1-59593-410-3. doi: http://doi.acm.org/10.1145/1138304.1138307.

Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9): 165–185, Oct 2007. ISSN 1660-1769. `http://www.jot.fm/contents/issue_2007_10/paper9.html`.

Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. Managing model conflicts in distributed development. In *MoDELS '08: 11th Internat. Conf. on Model Driven Engineering Languages and Systems*, pages 311–325, Berlin, Heidelberg, Oct 2008. Springer-Verlag. ISBN 978-3-540-87874-2. doi: http://dx.doi.org/10.1007/978-3-540-87875-9_23.

Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ives, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architecture: Views and Beyond*. SEI Series in Software Engineering. Pearson Education, Inc., Boston, MA, 2003. ISBN 0-201-70372-6.

Alistair Cockburn. *Writing Effective Use Cases*. Agile Software Development. Addison-Wesley, Boston, MA, USA, 2001. ISBN 0-201-70225-8.

Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Comput. Surv.*, 30(2):232–282, Jun 1998. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/280277.280280.

Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 3540311874.

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

EMF Ecore Package. Website of the Eclipse EMF project, Feb 2011. `http://download.eclipse.org/modeling/emf/emf/javadoc/2.6.0/org/eclipse/emf/e%core/package-summary.html#details`.

Jacky Estublier. Software configuration management: a roadmap. In *ICSE '00: Internat. Conf. on the Future of Software Engineering*, pages 279–289, New York, NY, USA, Jun 2000. ACM. ISBN 1-58113-253-0. doi: http://doi.acm.org/10.1145/336512.336576.

Jacky Estublier and Sergio Garcia. Process model and awareness in SCM. In *SCM '05: 12th Internat. Workshop on Software Configuration Management*, pages 59–74, New York, NY, USA, Sep 2005. ACM. ISBN 1-59593-310-7. doi: http://doi.acm.org/10.1145/1109128.1109133.

Jacky Estublier, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Trans. on Softw. Eng. and Methodol.*, 14(4):383–430, Oct 2005. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/1101815.1101817.

Franck Fleurey, Benoit Baudry, Robert France, and Sudipto Ghosh. A generic approach for automatic model composition. In *MoDELS '07: 11th Internat. Workshop on Aspect-Oriented Modeling*, pages 7–15, Berlin, Germany, Sep 2007. Springer-Verlag. ISBN 978-3-540-69069-6. doi: http://dx.doi.org/10.1007/978-3-540-69073-3_2.

Fujaba. Website of the Fujaba Tool Suite project, Nov 2009. `http://www.fujaba.de/home.html`.

Rebecca E. Grinter. Using a configuration management tool to coordinate software development. In *COCS '95: Conf. on Organizational Computing Systems*, pages 168–177, New York, NY, USA, Aug 1995. ACM. ISBN 0-89791-706-5. doi: http://doi.acm.org/10.1145/224019.224036.

Cecilia Haskins. Model driven development: Integrating tools with practices. In *ECBS '97: IEEE Internat. Conf. on the Engineering of Computer-Based Systems*, pages 421–426, Los Alamitos, CA, USA, Mar 1997. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/ECBS.1997.581923.

Petr Hnětynka and František Plášil. Distributed versioning model for MOF. In *WISICT '04: Winter Internat. Sympos. on Information and Communication Technologies*, pages 1–6. Trinity College Dublin, May 2004.

Udo Kelter, Jürgen Wehren, and Jörg Niere. A generic difference algorithm for UML models. In *SE '05, Fachtagung des GI-Fachbereichs Softwaretechnik*, pages 105–116. GI, Mar 2005. ISBN 3-88579-393-8.

Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter 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. ISBN 1-58113-383-9. doi: http://doi.acm.org/10.1145/383962.384020.

P. J. H. King. Conversion of decision tables to computer programs by rule mask techniques. *Commun. ACM*, 9(11):796–801, Nov 1966. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/365876.365896.

Maximilian Koegel, Jonas Helming, and Stephan 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. ISBN 978-1-4244-3714-6. doi: http://doi.ieeecomputersociety.org/10.1109/CVSM.2009.5071721.

Maximilian Koegel, Markus Herrmannsdoerfer, Yang Li, Jonas Helming, and Joern David. Comparing state- and operation-based change tracking on models. In *EDOC '10, 14th Internat. Enterprise Distributed Object Computing Conf.*, pages 163–172, Washington, DC, USA, Oct 2010. IEEE Computer Society. ISBN 978-0-7695-4163-1. doi: http://dx.doi.org/10.1109/EDOC.2010.15.

Dimitrios 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. ISBN 978-3-642-02673-7. doi: http://dx.doi.org/10.1007/978-3-642-02674-4_11.

Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. 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 2006a. ACM. ISBN 1-59593-410-3. doi: http://0-doi.acm.org.mercury.concordia.ca:80/10.1145/1138304.1138308.

Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Eclipse development tools for Epsilon. Eclipse Modeling Symposium, Eclipse Summit Europe 2006, Oct 2006b. `http://www.eclipsecon.org/summiteurope2006/presentations/ESE2006-EclipseModelingSymposium9_DevelopmentToolsforEpsilon.pdf`.

Dimitrios S. Kolovos, Louis Rose, Richard F. Paige, and Fiona A.C. Polack. The Epsilon Book. Website of the Epsilon subproject of the Eclipse GMT project, 2010. `http://www.eclipse.org/gmt/epsilon/doc/book`.

Tommy Lennhamn. Model driven development: Are MDD models assets or liabilities? *The Rational Edge*, Jul 2007. `http://www.ibm.com/developerworks/rational/library/jul07/lennhamn`.

Kim Letkeman. Comparing and merging UML models in IBM Rational Software Architect: Part 2. *IBM Developer Works*, Jul 2005a. `http://www-128.ibm.com/developerworks/rational/library/05/712_comp2`.

Kim Letkeman. Comparing and merging UML models in IBM Rational Software Architect: Part 3. *IBM Developer Works*, Aug 2005b. `http://www-128.ibm.com/developerworks/rational/library/05/802_comp3`.

Ernst Lippe and Norbert van Oosterom. Operation-based merging. *ACM SIGSOFT Softw. Eng. Notes*, 17(5):78–87, Dec 1992. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/142882.143753.

Boris Magnusson, Ulf Asklund, and Sten Minör. Fine-grained revision control for collaborative software development. In *SIGSOFT '93: 1st ACM SIGSOFT Sympos. on Foundations of Software Engineering*, pages 33–41, New York, NY, USA, Dec 1993. ACM. ISBN 0-89791-625-5. doi: http://doi.acm.org/10.1145/256428.167061.

Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Guest editors' introduction: Model-driven development. *IEEE Softw.*, 20(5):14–18, Sep 2003. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2003.1231145.

Tom Mens. A state-of-the-art survey on software merging. *IEEE Trans. on Softw. Eng.*, 28(5):449–462, May 2002. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2002.1000449.

Martin Monperrus. Ecore Metamodel, Dec 2010. `www.monperrus.net/martin/the-class-diagram-of-the-ecore-metamodel.pdf`.

Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. Parallel changes in large scale software development: an observational case study. *ACM Trans. on Softw. Eng. and Methodol.*, 10(3):308–337, Jul 2001. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/383876.383878.

Benjamin C. Pierce and Jérôme 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`.

Benjamin C. Pierce, Alan Schmitt, and Michael B. Greenwald. Bringing Harmony to optimism: an experiment in synchronizing heterogeneous tree-structured data. Technical Report MS-CIS-03-42, University of Pennsylvania, Philadelphia, PA, USA, Mar 2004. `http://www.cis.upenn.edu/~bcpierce/papers/harmony-sync-tr.pdf`.

Udo W. Pooch. Translation of decision tables. *ACM Comput. Surv.*, 6(2):125–151, Jun 1974. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/356628.356630.

Rachel A. Pottinger and Philip 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. ISBN 0-12-722442-4.

Yasushi Saito and Marc Shapiro. Replication: Optimistic approaches. Technical Report HPL-2002-33, Hewlett-Packard Laboratories, Mar 2002. `http://www.hpl.hp.com/techreports/2002/HPL-2002-33.pdf`.

Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Comput.*, 39(2):25–31, Feb 2006. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2006.58.

Maik Schmidt and Tilman Gloetzner. Constructing difference tools for models using the SiDiff framework. In *ICSE '08: Companion of the 30th Internat. Conf. on Software Engineering*, pages 947–948, New York, NY, USA, May 2008. ACM. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1370175.1370201.

Maik Schmidt, Sven Wenzel, Timo Kehrer, and Udo 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. ISBN 978-1-4244-3714-6. doi: http://doi.ieeecomputersociety.org/10.1109/CVSM.2009.5071716.

Christian Schneider. *CoObRA: Eine Plattform zur Verteilung und Replikation komplexer Objektstrukturen mit optimistischen Sperrkonzepten.* PhD thesis, University of Kassel, Kassel, Germany, Dec 2007.

Christian Schneider, Albert Zündorf, and Jörg Niere. CoObRA - a small step for development tools to collaborative environments. *IEEE Seminar Digests*, 2004 (902):21–28, 2004. doi: 10.1049/ic:20040206.

Ed Seidewitz. What models mean. *IEEE Softw.*, 20(5):26–32, Sep 2003. ISSN 0740-7459. doi: http://doi.ieeecomputersociety.org/10.1109/MS.2003.1231147.

Bran Selic. The pragmatics of model-driven development. *IEEE Softw.*, 20(5):19–25, Sep 2003. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2003.1231146.

Shane Sendall and Wojtek Kozaczynski. Model transformation: the heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, Sep 2003. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2003.1231150.

Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle, NJ, USA, 1996. ISBN 0-13-182957-2.

Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo 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. ISBN 978-1-59593-811-4. doi: http://doi.acm.org/10.1145/1287624.1287665.

Andrew Tridgell and Paul Mackerras. On-line man page for rsync, Feb 2011. `http://rsync.samba.org/ftp/rsync/rsync.html`.

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

Darcy Wiborg Weber. Change sets versus change packages: Comparing implementations of change-based SCM. In *ICSE '97: SCM-7 Workshop on System Configuration Management*, pages 25–35, London, UK, May 1997. Springer-Verlag. ISBN 3-540-63014-7.

Bernhard Westfechtel. A formal approach to three-way merging of EMF models. In *IWMCP '10: Internat. Workshop on Model Comparison in Practice*, pages 31–41, Jul 2010.

Zhenchang Xing and Eleni 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. ISBN 1-59593-993-4. doi: http://doi.acm.org/10.1145/1101908.1101919.

Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In *ASE '07, 22nd Internat. Conf. on Automated Software Engineering*, pages 164–173, New York, NY, USA, Nov 2007. ACM. ISBN 978-1-59593-882-4. doi: http://doi.acm.org/10.1145/1321631.1321657.

Appendices

Appendix A

Implementation Notes for the Hybrid Merge Workflow

The notes collected here provide implementation information concerning the respective phases of the hybrid workflow phases as described in Chapter 5.

Model Normalization

- Three model files are input for a three-way merge, and two for a two-way merge.
- The input files may be static (i.e., state-based), or dynamic (i.e., operation-based) representations of models.
- Upon input, models are normalized to an internal representation conforming to a common metamodel, suitable for state-based analysis and manipulation.
- The normal-form of the internal models decouples the workflow from modeling tools and change operation recorders.

Model Differencing

- A state-based difference is taken between normal-forms of the base model and its left and right replica-versions.
- Two state-based difference, or delta, models conforming to a difference metamodel are constructed as a result of the differencing.

- The difference metamodel extends the normal-form metamodel for capturing the evolution of the replica-versions.
- Unlike in the presentation of the workflow, in the implementation, the differencing phase occurs before the comparison phase.

Model Comparison

- The elements of the two state-based internal models are matched using a combination of default and possibly user-supplied matching strategies.
- Which matching strategies to employ are user-selectable, and the weights for combining their results user-settable.
- All matching decisions may be overridden by the user.
- Matching information from a previous merge session may be used to improve match accuracy and reduce the time spent correcting mismatches.
- Given the actual order of workflow phases, MMD elements will form part of the comparison, and will be available for matching.

Operation Extraction

- The difference models are used to construct a set of change operations for each side of the merge.
- The operations created, reflect the entities of the difference metamodel; that is, they either `Add`, `Delete` or `Alter` a model element.
- The recreation of operational traces transitions the workflow from the state-based to the operation-based approach.

Conflict Detection

- Changes are first ordered on the same side using the *before* predicate.
- Changes are then ordered *across* the sides, again with *before*: left-to-right and right-to-left. A conflict exists if the predicate is true in both directions.

- The three applications of *before* merge the models, possibly with conflicts.
- Decision tables are used to evaluate *before*.
- Table rules, conditions and actions are user-accessible and modifiable.

Conflict Resolution

- Automatic resolution of conflicts is also decision table-driven. Not all conflicts may be automatically resolvable.
- One last application of *before* gives the change operations of the merged transformation their final order, with conflicts moved as far down the list as possible.
- The final merged list of change operations is presented for user review and possible modification before execution.

Model Patching

- Patching allows for selective execution of change operations against the base model to produce a (partially) merged model in the normal-form.
- Patching will generate and catch an inconsistency error if user selections violate any change dependencies.

Model Denormalization

- This is the reverse of the Normalization phase.
- The conversion is from the normal-form merged model into the forms of the original input—one outcome for each type of input.
- The state-based merged model will be converted into an operation-based one if the input was of that form.
- Implementation of this phase is not complete, so currently, only Ecore models are outputted.

Appendix B

Mirador Batch Mode Use Case

There are two user-goal use cases that describe the merging behavior of Mirador: one for interactive mode, and one for non-interactive, or batch, mode. The main success scenario and extensions for the interactive merge were given in Subsection 6.2.2. The batch merge scenario and extensions appear on the next page.

Use Case: Run Batch Model Merge

Goal: To merge models with no input from the user

Primary Actor: Merger

Precondition: none

Main Success Scenario

1. Merger invokes Mirador with batch option, specifying desired arguments and any options.
2. System sets up merge based on passed arguments and options, loading specified models.
3. System matches model elements.
4. System detects and resolves conflicts producing a merged change set.
5. System patches base model with change set, and updates model display.
6. System outputs the (partially) merged model in the format(s) of the original[1].

   Use case ends successfully.

Extensions

1a. **A configuration file is specified on the command line.**

1a1. System parses configuration file arguments and options.

Use case continues.

2a. **A specified file is not found, or cannot be parsed.**

2a1. An error is displayed and logged.

Use case ends unsuccessfully.

2b. **The model format is not supported, or the model is not well formed.**

2b1. An error is displayed and logged.

Use case ends unsuccessfully.

4a. **Only two models were loaded.**

4a1. A 2-way merge is performed instead of a 3-way.

Use case continues.

5a. **An error is encountered applying a change.**

5a1. Change application halts, and the change is flagged as being in error.

Use case continues.

Postcondition: Merged model and merge report have been stored.

[1]Not yet supported. Currently, all models are output as Ecore XMI files regardless of input type.

Appendix C

Mirador Command Line and Configuration File Syntax

Unless passed the `-b` batch mode option, Mirador always starts in interactive mode which will bring up the Model Input Panel. If none of the other options, and none of the arguments of Figure 70 are passed, the fields and controls of the GUI will either be empty, or in their default state.

The display of usage help is geared towards command line invocation, but anything shown, with the exception of the `-c` option, may be placed in a configuration file, an example of which may be seen in Figure 71. The format is free form with the only restriction being that the model arguments must be listed in the order as given in the syntax description.

```
Usage: java Mirador [OPTION] [[BASE_MODEL] LEFT_MODEL RIGHT_MODEL]
Note: BASE_MODEL is required for three-way merging.

General options:
 -?              display this help and exit
 -v              print version information and exit
 -c=FILE         specify configuration file
 -d              enable printing of debug messages

Element matching:
 -ml             select left as master side
 -mr             select right as master side
 -mt=VAL         set element matching threshold
 -mp=FILE        specify previous match file
 -md[=VAL]       use 'by dependency' matching strategy
 -me[=VAL]       use 'by ECL' matching strategy
 -mi[=VAL]       use 'by ID' matching strategy
 -mn[=VAL]       use 'by name' matching strategy
 -ms[=VAL]       use 'by structure' matching strategy
 -m1=FILE[,VAL]  use 'by user #1' matching strategy
 -m2=FILE[,VAL]  use 'by user #2' matching strategy
 -mt=VAL         set element matching threshold

Model merging:
 -b              run in batch mode
 -i              run in interactive mode - default
 -td=FILE        specify conflict detection decision table
 -tr=FILE        specify conflict resolution decision table
```

Figure 70: Mirador usage help

```
// Mirador configuration file to load "HTML Document".

// Options:
-d -mi=0.4 -mn=0.8 -ms=0.4 -md -me=0.5 -m1=UserEvaluator1,0.6 -mt=0.6
-tc=usr/table-before.ddf -tr=usr/table-resolve.ddf

// Arguments:
../../models/html/html.ctr
../../models/html/lf/html.ctr
../../models/html/rt/html.ctr
```

Figure 71: Mirador configuration file for “HTML Document” example

Appendix D

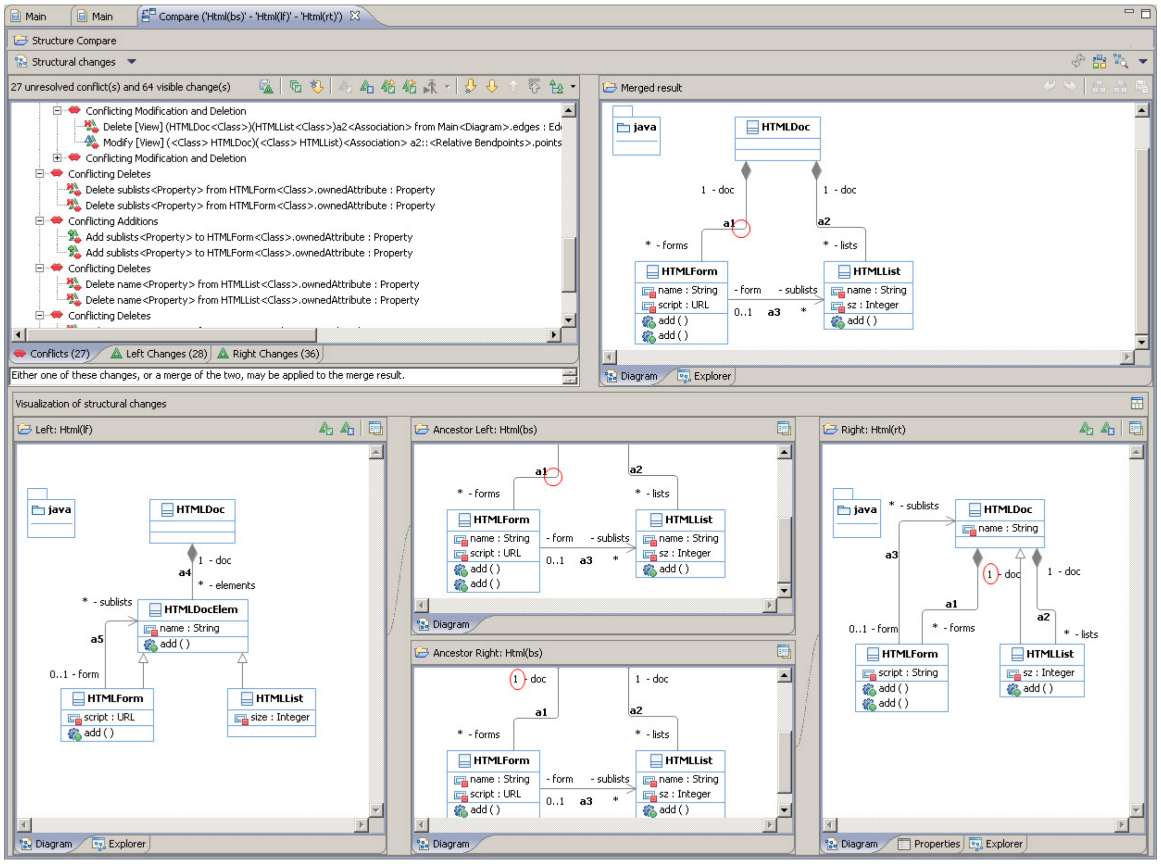
Running Merge Example in IBM Rational Software Architect

This appendix collects a number of screenshots of the "HTML Document" example, which was used for Mirador validation, being merged in RSA 8.01.

Figure 72 shows the three example models being compared in RSA, with the base model at the top, and the left and right versions on their respective sides. The two panes between the replica-versions show the ancestor for each replica, which in a three-way merge is the same as the base model.

All the changes found by RSA, on either side of the merge, are pictured in Figures 73 and 74. Conflicting changes have been expanded to show their counterpart on the opposite side.

The detected conflicts for the merge are shown in Figure 75. The five expanded conflicts at the bottom of the figure are the only ones that are related to model changes, which is all Mirador is concerned with. The remainder (three of which have been expanded) have to do with changes to the view of the class diagram.

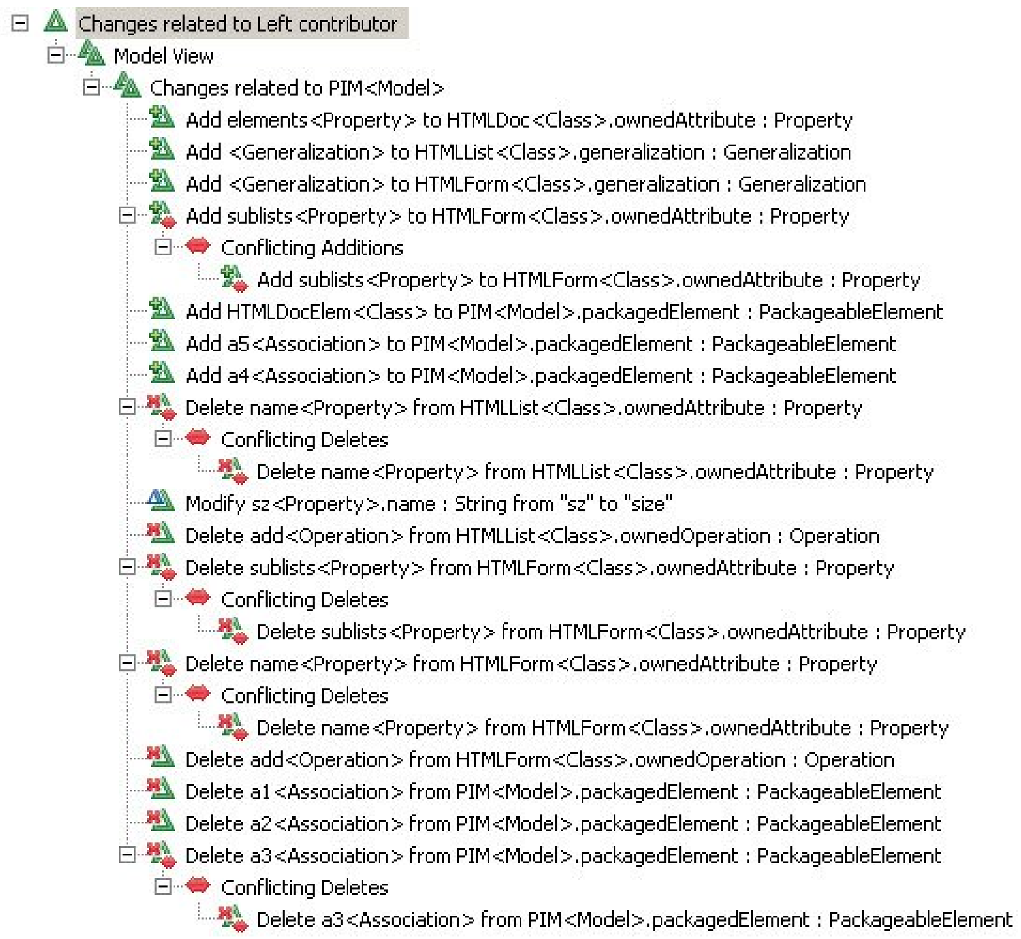


27 unresolved conflict(s) and 64 visible change(s)

```
Conflicting Modification and Deletion
    Delete [View] (HTMLDoc<Class>)(HTMLList<Class>)a2<Association> from Main<Diagram>.edges : Ed
    Modify [View] (<Class> HTMLDoc)(<Class> HTMLList)<Association> a2::<Relative Bendpoints>.points
Conflicting Modification and Deletion
Conflicting Deletes
    Delete sublists<Property> from HTMLForm<Class>.ownedAttribute : Property
    Delete sublists<Property> from HTMLForm<Class>.ownedAttribute : Property
Conflicting Additions
    Add sublists<Property> to HTMLForm<Class>.ownedAttribute : Property
    Add sublists<Property> to HTMLForm<Class>.ownedAttribute : Property
Conflicting Deletes
    Delete name<Property> from HTMLList<Class>.ownedAttribute : Property
    Delete name<Property> from HTMLList<Class>.ownedAttribute : Property
Conflicting Deletes
```

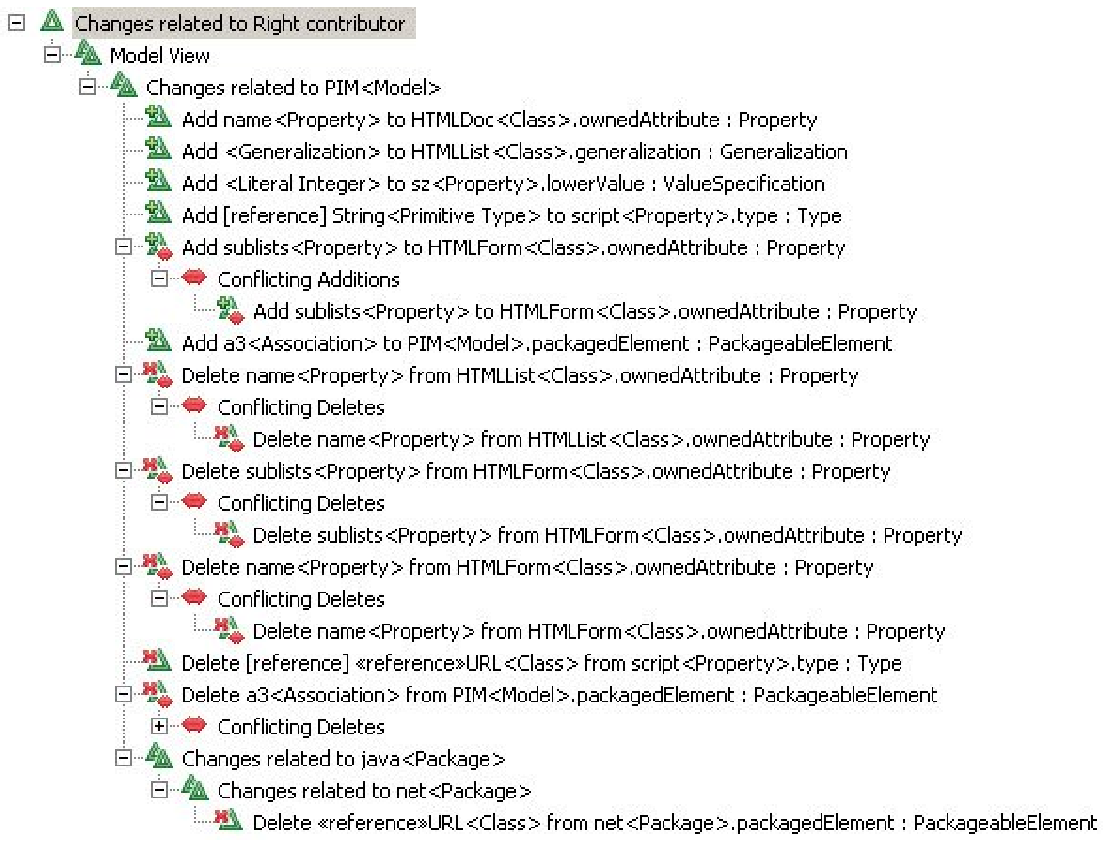
Either one of these changes, or a merge of the two, may be applied to the merge result.

Figure 72: "HTML Document" three-way compare in Rational Software Architect



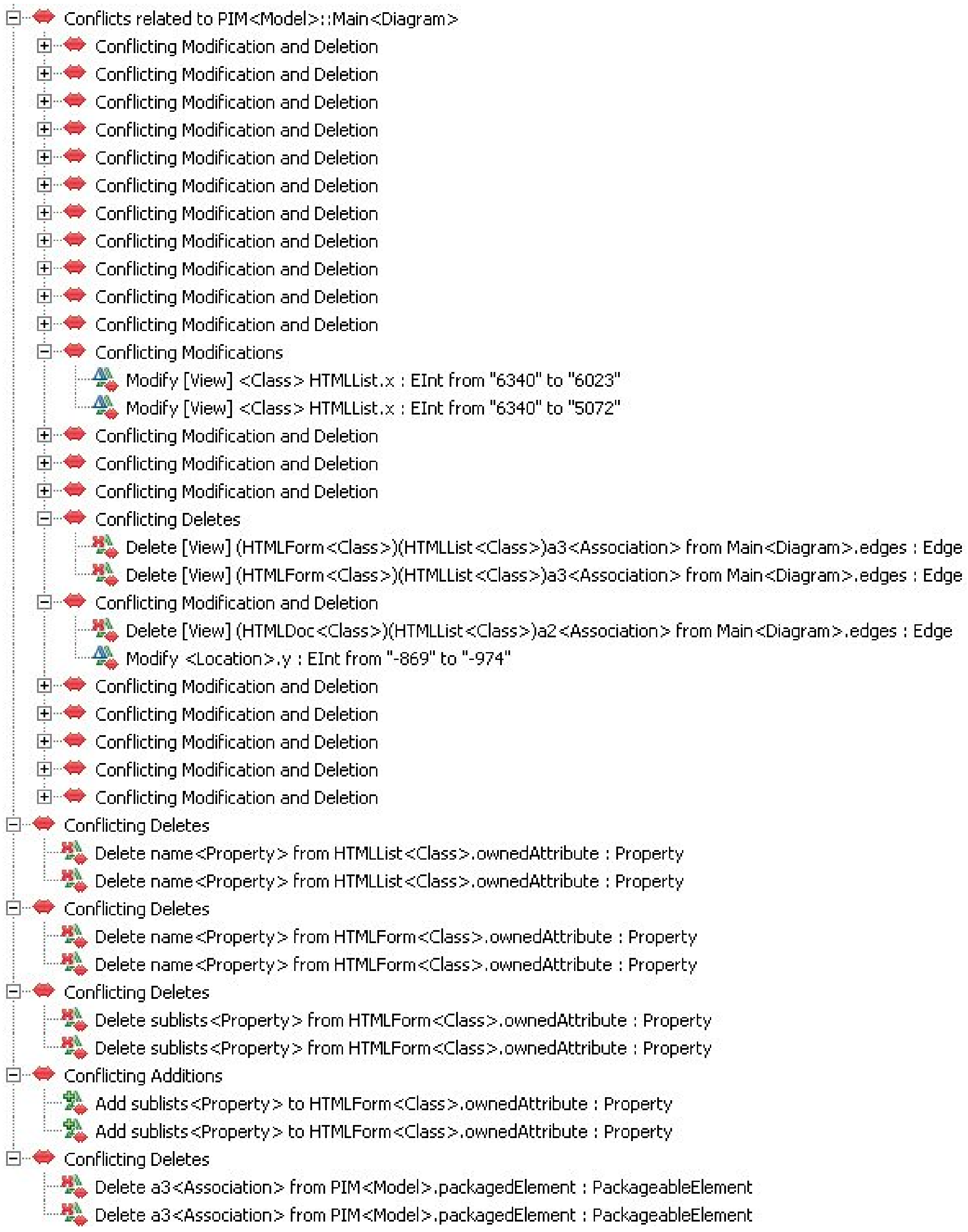
```
Changes related to Left contributor
  Model View
    Changes related to PIM<Model>
      Add elements<Property> to HTMLDoc<Class>.ownedAttribute : Property
      Add <Generalization> to HTMLList<Class>.generalization : Generalization
      Add <Generalization> to HTMLForm<Class>.generalization : Generalization
      Add sublists<Property> to HTMLForm<Class>.ownedAttribute : Property
        Conflicting Additions
          Add sublists<Property> to HTMLForm<Class>.ownedAttribute : Property
      Add HTMLDocElem<Class> to PIM<Model>.packagedElement : PackageableElement
      Add a5<Association> to PIM<Model>.packagedElement : PackageableElement
      Add a4<Association> to PIM<Model>.packagedElement : PackageableElement
      Delete name<Property> from HTMLList<Class>.ownedAttribute : Property
        Conflicting Deletes
          Delete name<Property> from HTMLList<Class>.ownedAttribute : Property
      Modify sz<Property>.name : String from "sz" to "size"
      Delete add<Operation> from HTMLList<Class>.ownedOperation : Operation
      Delete sublists<Property> from HTMLForm<Class>.ownedAttribute : Property
        Conflicting Deletes
          Delete sublists<Property> from HTMLForm<Class>.ownedAttribute : Property
      Delete name<Property> from HTMLForm<Class>.ownedAttribute : Property
        Conflicting Deletes
          Delete name<Property> from HTMLForm<Class>.ownedAttribute : Property
      Delete add<Operation> from HTMLForm<Class>.ownedOperation : Operation
      Delete a1<Association> from PIM<Model>.packagedElement : PackageableElement
      Delete a2<Association> from PIM<Model>.packagedElement : PackageableElement
      Delete a3<Association> from PIM<Model>.packagedElement : PackageableElement
        Conflicting Deletes
          Delete a3<Association> from PIM<Model>.packagedElement : PackageableElement
```

Figure 73: Left “HTML Document” changes in Rational Software Architect



```
Changes related to Right contributor
  Model View
    Changes related to PIM<Model>
      Add name<Property> to HTMLDoc<Class>.ownedAttribute : Property
      Add <Generalization> to HTMLList<Class>.generalization : Generalization
      Add <Literal Integer> to sz<Property>.lowerValue : ValueSpecification
      Add [reference] String<Primitive Type> to script<Property>.type : Type
      Add sublists<Property> to HTMLForm<Class>.ownedAttribute : Property
        Conflicting Additions
          Add sublists<Property> to HTMLForm<Class>.ownedAttribute : Property
      Add a3<Association> to PIM<Model>.packagedElement : PackageableElement
      Delete name<Property> from HTMLList<Class>.ownedAttribute : Property
        Conflicting Deletes
          Delete name<Property> from HTMLList<Class>.ownedAttribute : Property
      Delete sublists<Property> from HTMLForm<Class>.ownedAttribute : Property
        Conflicting Deletes
          Delete sublists<Property> from HTMLForm<Class>.ownedAttribute : Property
      Delete name<Property> from HTMLForm<Class>.ownedAttribute : Property
        Conflicting Deletes
          Delete name<Property> from HTMLForm<Class>.ownedAttribute : Property
      Delete [reference] «reference»URL<Class> from script<Property>.type : Type
      Delete a3<Association> from PIM<Model>.packagedElement : PackageableElement
        Conflicting Deletes
      Changes related to java<Package>
        Changes related to net<Package>
          Delete «reference»URL<Class> from net<Package>.packagedElement : PackageableElement
```

Figure 74: Right “HTML Document” changes in Rational Software Architect



```
Conflicts related to PIM<Model>::Main<Diagram>
    Conflicting Modification and Deletion
    Conflicting Modification and Deletion
    Conflicting Modification and Deletion
    Conflicting Modification and Deletion
    Conflicting Modification and Deletion
    Conflicting Modification and Deletion
    Conflicting Modification and Deletion
    Conflicting Modification and Deletion
    Conflicting Modification and Deletion
    Conflicting Modification and Deletion
    Conflicting Modification and Deletion
    Conflicting Modifications
        Modify [View] <Class> HTMLList.x : EInt from "6340" to "6023"
        Modify [View] <Class> HTMLList.x : EInt from "6340" to "5072"
    Conflicting Modification and Deletion
    Conflicting Modification and Deletion
    Conflicting Modification and Deletion
    Conflicting Deletes
        Delete [View] (HTMLForm<Class>)(HTMLList<Class>)a3<Association> from Main<Diagram>.edges : Edge
        Delete [View] (HTMLForm<Class>)(HTMLList<Class>)a3<Association> from Main<Diagram>.edges : Edge
    Conflicting Modification and Deletion
        Delete [View] (HTMLDoc<Class>)(HTMLList<Class>)a2<Association> from Main<Diagram>.edges : Edge
        Modify <Location>.y : EInt from "-869" to "-974"
    Conflicting Modification and Deletion
    Conflicting Modification and Deletion
    Conflicting Modification and Deletion
    Conflicting Modification and Deletion
    Conflicting Modification and Deletion
Conflicting Deletes
    Delete name<Property> from HTMLList<Class>.ownedAttribute : Property
    Delete name<Property> from HTMLList<Class>.ownedAttribute : Property
Conflicting Deletes
    Delete name<Property> from HTMLForm<Class>.ownedAttribute : Property
    Delete name<Property> from HTMLForm<Class>.ownedAttribute : Property
Conflicting Deletes
    Delete sublists<Property> from HTMLForm<Class>.ownedAttribute : Property
    Delete sublists<Property> from HTMLForm<Class>.ownedAttribute : Property
Conflicting Additions
    Add sublists<Property> to HTMLForm<Class>.ownedAttribute : Property
    Add sublists<Property> to HTMLForm<Class>.ownedAttribute : Property
Conflicting Deletes
    Delete a3<Association> from PIM<Model>.packagedElement : PackageableElement
    Delete a3<Association> from PIM<Model>.packagedElement : PackageableElement
```

Figure 75: “HTML Document” conflicts in Rational Software Architect

Appendix E

Model Files of Running Merge Example

To get a feel for the differences between state-based and operation-based model types, the appendix presents a selection of listings of the various model files that make up the “HTML Document” running example. In order to facilitate fit, it was necessary to use a small font size for the listings. For reference, the Ecore metamodel is also reproduced here [Monperrus, 2010].

Being of a reasonable size, the Ecore XMI files for all three versions of the example model are given in their entirety. The Fujaba models, however, are rather large—the base model file is 445 lines long, and the replica-versions, since they contain the base model, are even longer. For this reason, only snippets from the three versions of the model are shown. The syntax for the CoObRA log records that make up a Fujaba model file can be found in Appendix F.

E.1 The Ecore Metamodel

Figure 76 is a class diagram for the state-based Ecore metamodel, showing its meta-classes and their relationships. The metamodel defines the normal-form of the hybrid merge workflow.

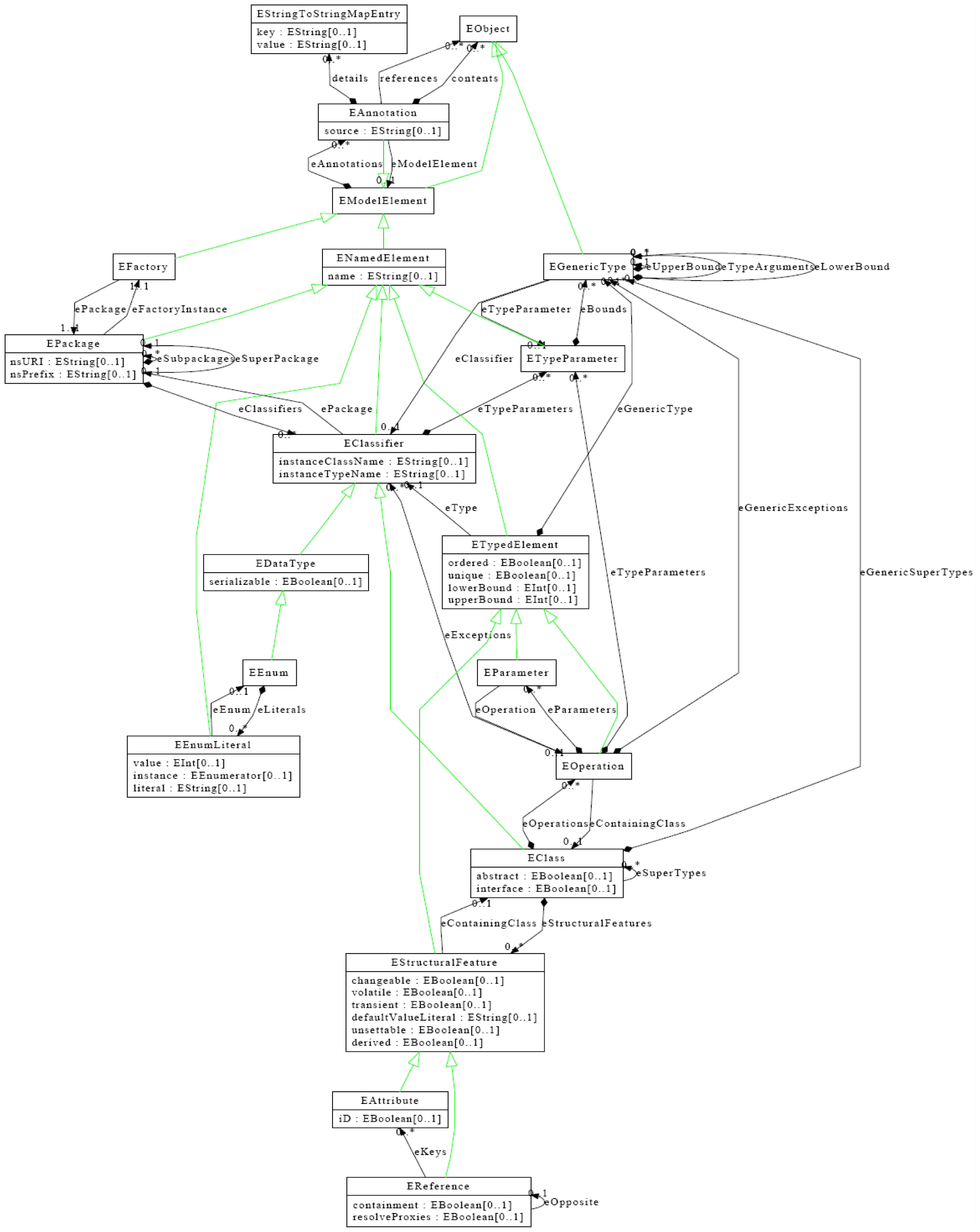


Figure 76: Class Diagram of the Ecore Metamodel

E.2 Ecore XMI Models

Below is the Ecore XMI for the base model of the “HTML Document” example. The various EElement entities are Ecore metaclasses which are used to represent UML elements of the model. The representation contains only state, no operations.

```
<?xml version="1.0" encoding="ASCII"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ecore=
    "http://www.eclipse.org/emf/2002/Ecore" xmi:id="qyd2E_92" name="html-doc">
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_2" name="Boolean" instanceClassName="java.lang.Boolean"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_3" name="String" instanceClassName="java.lang.String"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_4" name="Integer" instanceClassName="java.lang.Integer"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_5" name="Byte" instanceClassName="java.lang.Byte"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_8" name="Float" instanceClassName="java.lang.Float"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_9" name="Double" instanceClassName="java.lang.Double"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_A" name="Character" instanceClassName="java.lang.Character"/>
  <eClassifiers xsi:type="ecore:EClass" xmi:id="qyd2E_E2" name="HTMLDoc">
    <eStructuralFeatures xsi:type="ecore:EReference" xmi:id="WnHJn_291" name="forms" upperBound="-1" eType="qyd2E_63"
        containment="true" eOpposite="WnHJn_191"/>
    <eStructuralFeatures xsi:type="ecore:EReference" xmi:id="YXqWO_6F1" name="lists" upperBound="-1" eType="qyd2E_E3"
        containment="true" eOpposite="YXqWO_5F1"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" xmi:id="qyd2E_63" name="HTMLForm">
    <eOperations xmi:id="qyd2E_94" name="add">
      <eParameters xmi:id="qyd2E_A4" name="to_add" eType="qyd2E_3"/>
    </eOperations>
    <eOperations xmi:id="qyd2E_D4" name="add">
      <eParameters xmi:id="qyd2E_E4" name="to_add" eType="qyd2E_E3"/>
    </eOperations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" xmi:id="qyd2E_35" name="name" eType="qyd2E_3"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" xmi:id="DJJ3S_661" name="script" eType="McEU4_76"/>
    <eStructuralFeatures xsi:type="ecore:EReference" xmi:id="WnHJn_191" name="doc" eType="qyd2E_E2" eOpposite="WnHJn_291"/>
    <eStructuralFeatures xsi:type="ecore:EReference" xmi:id="WN3I6_223" name="sublists" upperBound="-1" eType="qyd2E_E3"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" xmi:id="qyd2E_E3" name="HTMLList">
    <eOperations xmi:id="qyd2E_05" name="add">
      <eParameters xmi:id="qyd2E_15" name="to_add" eType="qyd2E_3"/>
    </eOperations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" xmi:id="DJJ3S_861" name="name" eType="qyd2E_3"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" xmi:id="DJJ3S_961" name="sz" eType="qyd2E_4"/>
    <eStructuralFeatures xsi:type="ecore:EReference" xmi:id="YXqWO_5F1" name="doc" eType="qyd2E_E2" eOpposite="YXqWO_6F1"/>
  </eClassifiers>
  <eSubpackages xmi:id="McEU4_96" name="java">
    <eSubpackages xmi:id="McEU4_A6" name="net">
      <eClassifiers xsi:type="ecore:EClass" xmi:id="McEU4_76" name="URL"/>
    </eSubpackages>
  </eSubpackages>
</ecore:EPackage>
```

Below is the Ecore XMI of the left model.

```
<?xml version="1.0" encoding="ASCII"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ecore=
    "http://www.eclipse.org/emf/2002/Ecore" xmi:id="qyd2E_92" name="html-doc">
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_2" name="Boolean" instanceClassName="java.lang.Boolean"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_3" name="String" instanceClassName="java.lang.String"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_4" name="Integer" instanceClassName="java.lang.Integer"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_5" name="Byte" instanceClassName="java.lang.Byte"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_8" name="Float" instanceClassName="java.lang.Float"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_9" name="Double" instanceClassName="java.lang.Double"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_A" name="Character" instanceClassName="java.lang.Character"/>
  <eClassifiers xsi:type="ecore:EClass" xmi:id="qyd2E_E2" name="HTMLDoc">
    <eStructuralFeatures xsi:type="ecore:EReference" xmi:id="Fu6CN_B15" name="elements" upperBound="-1" eType="sCE3W_4B4"
      containment="true" eOpposite="Fu6CN_A15"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" xmi:id="qyd2E_63" name="HTMLForm" eSuperTypes="sCE3W_4B4">
    <eOperations xmi:id="qyd2E_D4" name="add">
      <eParameters xmi:id="qyd2E_E4" name="to_add" eType="qyd2E_E3"/>
    </eOperations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" xmi:id="DJJ3S_661" name="script" eType="McEU4_76"/>
    <eStructuralFeatures xsi:type="ecore:EReference" xmi:id="Fu6CN_D25" name="sublists" upperBound="-1" eType="sCE3W_4B4"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" xmi:id="qyd2E_E3" name="HTMLList" eSuperTypes="sCE3W_4B4">
    <eStructuralFeatures xsi:type="ecore:EAttribute" xmi:id="DJJ3S_961" name="size" eType="qyd2E_4"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" xmi:id="sCE3W_4B4" name="HTMLDocElem">
    <eOperations xmi:id="sCE3W_0C4" name="add">
      <eParameters xmi:id="sCE3W_1C4" name="to_add" eType="qyd2E_3"/>
    </eOperations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" xmi:id="sCE3W_DB4" name="name" eType="qyd2E_3"/>
    <eStructuralFeatures xsi:type="ecore:EReference" xmi:id="Fu6CN_A15" name="doc" eType="qyd2E_E2" eOpposite="Fu6CN_B15"/>
  </eClassifiers>
  <eSubpackages xmi:id="McEU4_96" name="java">
    <eSubpackages xmi:id="McEU4_A6" name="net">
      <eClassifiers xsi:type="ecore:EClass" xmi:id="McEU4_76" name="URL"/>
    </eSubpackages>
  </eSubpackages>
</ecore:EPackage>
```

And lastly, the Ecore XMI of the right model.

```
<?xml version="1.0" encoding="ASCII"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ecore=
    "http://www.eclipse.org/emf/2002/Ecore" xmi:id="qyd2E_92" name="html-doc">
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_2" name="Boolean" instanceClassName="java.lang.Boolean"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_3" name="String" instanceClassName="java.lang.String"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_4" name="Integer" instanceClassName="java.lang.Integer"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_5" name="Byte" instanceClassName="java.lang.Byte"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_8" name="Float" instanceClassName="java.lang.Float"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_9" name="Double" instanceClassName="java.lang.Double"/>
  <eClassifiers xsi:type="ecore:EDataType" xmi:id="qyd2E_A" name="Character" instanceClassName="java.lang.Character"/>
  <eClassifiers xsi:type="ecore:EClass" xmi:id="qyd2E_E2" name="HTMLDoc">
    <eStructuralFeatures xsi:type="ecore:EAttribute" xmi:id="DHxBH_9" name="name" eType="qyd2E_3"/>
    <eStructuralFeatures xsi:type="ecore:EReference" xmi:id="WnHJn_291" name="forms" upperBound="-1" eType="qyd2E_63"
      containment="true" eOpposite="WnHJn_191"/>
    <eStructuralFeatures xsi:type="ecore:EReference" xmi:id="YXqW0_6F1" name="lists" upperBound="-1" eType="qyd2E_E3"
      containment="true" eOpposite="YXqW0_5F1"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" xmi:id="qyd2E_63" name="HTMLForm">
    <eOperations xmi:id="qyd2E_94" name="add">
      <eParameters xmi:id="qyd2E_A4" name="to_add" eType="qyd2E_3"/>
    </eOperations>
    <eOperations xmi:id="qyd2E_D4" name="add">
      <eParameters xmi:id="qyd2E_E4" name="to_add" eType="qyd2E_E3"/>
    </eOperations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" xmi:id="DJJ3S_661" name="script" eType="qyd2E_3"/>
    <eStructuralFeatures xsi:type="ecore:EReference" xmi:id="WnHJn_191" name="doc" eType="qyd2E_E2" eOpposite="WnHJn_291"/>
    <eStructuralFeatures xsi:type="ecore:EReference" xmi:id="DHxBH_3" name="sublists" upperBound="-1" eType="qyd2E_E2"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" xmi:id="qyd2E_E3" name="HTMLList">
    <eOperations xmi:id="qyd2E_05" name="add">
      <eParameters xmi:id="qyd2E_15" name="to_add" eType="qyd2E_3"/>
    </eOperations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" xmi:id="DJJ3S_961" name="sz" lowerBound="1" eType="qyd2E_4"/>
    <eStructuralFeatures xsi:type="ecore:EReference" xmi:id="YXqW0_5F1" name="doc" eType="qyd2E_E2" eOpposite="YXqW0_6F1"/>
  </eClassifiers>
  <eSubpackages xmi:id="McEU4_96" name="java">
    <eSubpackages xmi:id="McEU4_A6" name="net"/>
  </eSubpackages>
</ecore:EPackage>
```

E.3 Fujaba CoObRA Model Snippets

Fujaba models are operation-based, therefore, a model file consists of a sequence of change operation records. State is not directly represented, but may be extracted for any point in time from the operations in the sequence. The opening CoObRA records of the Fujaba base model file for the “HTML Document” example are listed below.

Included are some prelude header records, then the creation of basic data types (only partially shown) and various diagram elements such as cardinalities and stereotypes (not shown). And finally, near the bottom, the creation of the root "html-doc" package, and the class diagram "ClassDiagram1."

```
h;CoObRA2 Change stream - Version ;0;3
h;Encoding;UTF-8
h;ApplicationModel;Fujaba
h;Application Name;Fujaba Tool Suite
h;Fujaba Version;5.2.2
h;reuseFAM;true
t;qyd2E#;newUMLProject;1287172487209;-;;
c1;;v::de.uni_paderborn.fujaba.uml.common.UMLProject;i:qyd2E#1;-;i:qyd2E#;
c1;;v::de.uni_paderborn.fujaba.uml.structure.UMLBaseType;i:qyd2E#2;v:java.lang.String:Boolean;i:qyd2E#;
c3;;i:qyd2E#2;name;v::Boolean;-;-;i:qyd2E#;
c3;;i:qyd2E#2;progLangType;v::Boolean;-;-;i:qyd2E#;
c3;;i:qyd2E#2;progLangType;v::boolean;v::Boolean;-;i:qyd2E#;
c1;;v::de.uni_paderborn.fujaba.uml.structure.UMLBaseType;i:qyd2E#3;v:java.lang.String:String;i:qyd2E#;
c3;;i:qyd2E#3;name;v::String;-;-;i:qyd2E#;
c3;;i:qyd2E#3;progLangType;v::String;-;-;i:qyd2E#;
```

•
•
•

```
c1;;v::de.uni_paderborn.fujaba.uml.structure.UMLPackage;i:qyd2E#92;-;i:qyd2E#;
c3;;i:qyd2E#1;rootPackage;i:qyd2E#92;-;-;i:qyd2E#;
c3;;i:qyd2E#1;name;v::html-doc;-;-;i:qyd2E#;
t;qyd2E#A2;newClassDiagram;1287172504964;-;;
c1;;v::de.uni_paderborn.fujaba.uml.structure.UMLClassDiagram;i:qyd2E#B2;-;i:qyd2E#A2;
c3;;i:qyd2E#1;modelRootNodes;i:qyd2E#B2;-;-;i:qyd2E#A2;
c3;;i:qyd2E#B2;name;v::ClassDiagram1;-;-;i:qyd2E#A2;
```

The following snippet shows the creation of class **HTMLDoc** in the base model. In addition to change operations for creating the class in the model, there are others to put it into the correct package, diagram, and file, complete with back references.

```
t;qyd2E#D2;editClass;1287172549797;-;;
c1;;v::de.uni_paderborn.fujaba.uml.structure.UMLClass;i:qyd2E#E2;-;i:qyd2E#D2;
c3;;i:qyd2E#E2;declaredInPackage;i:qyd2E#92;-;-;i:qyd2E#D2;
c1;;v::de.uni_paderborn.fujaba.uml.common.UMLFile;i:qyd2E#F2;-;i:qyd2E#D2;
c3;;i:qyd2E#F2;contains;i:qyd2E#E2;-;-;i:qyd2E#D2;
c3;;i:qyd2E#E2;file;i:qyd2E#F2;-;-;i:qyd2E#D2;
c3;;i:qyd2E#E2;declaredInPackage;-;i:qyd2E#92;-;i:qyd2E#D2;
c3;;i:qyd2E#F2;contains;-;i:qyd2E#E2;-;i:qyd2E#D2;
c3;;i:qyd2E#E2;file;-;i:qyd2E#F2;-;i:qyd2E#D2;
c3;;i:qyd2E#E2;name;v::HTMLDoc;-;-;i:qyd2E#D2;
c3;;i:qyd2E#E2;declaredInPackage;i:qyd2E#92;-;-;i:qyd2E#D2;
c3;;i:qyd2E#F2;contains;i:qyd2E#E2;-;-;i:qyd2E#D2;
c3;;i:qyd2E#E2;file;i:qyd2E#F2;-;-;i:qyd2E#D2;
c3;;i:qyd2E#F2;name;v::HTMLDoc;-;-;i:qyd2E#D2;
c3;;i:qyd2E#B2;elements;i:qyd2E#E2;-;-;i:qyd2E#D2;
```

The creation of association **a4** in the left model is more complicated, with three elements being required to represent the bidirectional link between **HTMLDoc** and **HTMLDocElem**. For conversion to the normal-form, this structure must be mapped to two **EReference** objects.

```
t;Fu6CN#915;editAssoc;1287613457505;-;;
c1;;v::de.uni_paderborn.fujaba.uml.structure.UMLRole;i:Fu6CN#A15;-;i:Fu6CN#915;
c1;;v::de.uni_paderborn.fujaba.uml.structure.UMLRole;i:Fu6CN#B15;-;i:Fu6CN#915;
c1;;v::de.uni_paderborn.fujaba.uml.structure.UMLAssoc;i:Fu6CN#C15;-;i:Fu6CN#915;
c3;;i:Fu6CN#A15;name;v::doc;-;-;i:Fu6CN#915;
c3;;i:Fu6CN#B15;name;v::elements;-;-;i:Fu6CN#915;
c3;;i:Fu6CN#C15;name;v::a4;-;-;i:Fu6CN#915;
c3;;i:Fu6CN#A15;revLeftRole;i:Fu6CN#C15;-;-;i:Fu6CN#915;
c3;;i:Fu6CN#C15;leftRole;i:Fu6CN#A15;-;-;i:Fu6CN#915;
c3;;i:Fu6CN#B15;revRightRole;i:Fu6CN#C15;-;-;i:Fu6CN#915;
c3;;i:Fu6CN#C15;rightRole;i:Fu6CN#B15;-;-;i:Fu6CN#915;
c3;;i:Fu6CN#A15;adornment;v::2;v::0;-;i:Fu6CN#915;
c3;;i:Fu6CN#A15;card;i:WnHJn#491;-;-;i:Fu6CN#915;
c3;;i:qyd2E#E2;roles;i:Fu6CN#A15;-;-;i:Fu6CN#915;
c3;;i:Fu6CN#A15;target;i:qyd2E#E2;-;-;i:Fu6CN#915;
c3;;i:Fu6CN#B15;card;i:WnHJn#591;-;-;i:Fu6CN#915;
c3;;i:sCE3W#4B4;roles;i:Fu6CN#B15;-;-;i:Fu6CN#915;
c3;;i:Fu6CN#B15;target;i:sCE3W#4B4;-;-;i:Fu6CN#915;
c3;;i:qyd2E#B2;elements;i:Fu6CN#C15;-;-;i:Fu6CN#915;
c3;;i:Fu6CN#C15;diagrams;i:qyd2E#B2;-;-;i:Fu6CN#915;
```

The last snippet is taken from the right model. It shows the altering of the data type of attribute **script** (ID = DJJ3S#661) from **URL** (ID = McEU4#76) to **String** (ID = qyd2E#3), and the deleting of attribute **name** (ID = qyd2E#35) from class **HTMLForm** (ID = qyd2E#63).

```
t;DHxBH#C;inplace editing;1297992361367;-;;
c3;;i:DJJ3S#661;attrType;i:qyd2E#3;i:McEU4#76;-;i:DHxBH#C;
c3;;i:qyd2E#63;attrs;-;i:qyd2E#35;-;i:DHxBH#C;
c3;;i:qyd2E#35;parent;-;i:qyd2E#63;-;i:DHxBH#C;
c3;;i:qyd2E#35;attrType;-;i:qyd2E#3;-;i:DHxBH#C;
c2;;i:qyd2E#35;-;-;-;-;i:DHxBH#C;
```

Appendix F

CoObRA Change Record Grammar

As already stated, Fujaba models are actually logs of change operations maintained by the CoObRA (Concurrent Object Replication frAmework) versioning software. CoObRA tracks model changes as a sequence of file records grouped into transactions. The file records consist of semicolon-delimited fields.

The production rules for the CoObRA grammar are described herein using a variation of EBNF. The syntax that follows cannot be considered complete: It was possible to derive only some of the syntax from the designer's Ph.D. thesis [Schneider, 2007]; the rest had to be deduced from reading the source code and by conducting experiments through Fujaba.

F.1 Constants and Enumerations

The Java definitions of this section map between the text of the model files and the literals of the production rules.

Record Type Flag Constants

Each CoObRA record begins with a field containing a predefined flag character that identifies the record's type. All are single character fields, except for the change type ('c'), which includes another character identifying the change's kind.

LINE_MARKER_HEADER = ‘h’
LINE_MARKER_COMMENT = ‘#’
LINE_MARKER_TRANSACTION = ‘t’
LINE_MARKER_CHANGE = ‘c’
LINE_MARKER_CHANGE_NEXT = ‘x’
LINE_MARKER_CHANGE_PREVIOUS = ‘v’
LINE_MARKER_CHANGE_UNDONE = ‘u’
LINE_MARKER_CONFLICT_START = ‘<’
LINE_MARKER_CONFLICT_MID = ‘=’
LINE_MARKER_CONFLICT_END = ‘>’
LINE_MARKER_EOF = ‘]’
SEPARATOR_CHAR = ‘;’

Change Kind Enumeration

The second character of a change record type field (‘c’) must be one of these enumerated values:

UNDEFINED = 0	The type of change was not set.
CREATE_OBJECT = 1	A new object was added to the repository.
DESTROY_OBJECT = 2	An existing object was removed from the repository.
ALTER_FIELD = 3	A field was altered—remove old value, insert new one.
REMOVE_KEY = 4	A key was removed from a qualified field.
MANAGE = 5	Management data have changed (named objects, IDs, etc.)

Modifier Enumeration

These are hexadecimal characters to represent the source of a modification. They are applicable to transaction entries that include changes as part of the transaction:

MODIFIER_INVALID = 0x00	Invalid entry, not processed.
MODIFIER_TRANSIENT = 0x01	Temporary entry, ignored.
MODIFIER_LOCAL = 0x04	Local entry, not sent to server.
MODIFIER_SERVER = 0x08	Server entry, not resent to server.
MODIFIER_DEFAULT = 0x10	Default entry, processed normally.
MODIFIER_APPLICATION_OFFSET = 0x100	Application specific filters.

F.2 Production Rules

CoObRA logs changes to the artifact under its supervision with the various types of log records described here. The logging is fine-grained, so a seemingly simple change can produce many records. For versioning purposes, changes are encapsulated within transactions, which may, in turn, be nested.

Fujaba shadows user modeling with the construction of an abstract syntax graph (ASG). The actions associated with this graphical model are also recorded in the CoObRA change log. The actions are primarily concerned with properties for the visual presentation of the shadowed model elements.

Record Types

record

comment **\n**

header **\n**

transaction **\n**

change **\n**

undo **\n**

conflict **\n**

Comment Record

comment

# ; string

Header Record

header

h ; *metadata*

metadata

string (; string)*

Undo Record

undo [LINE_MARKER_CHANGE_UNDONE]

u **;** *undo-body*

Change Record

change

c *ch-kind* **;** *modifier* **;** *ch-body* **;**

ch-kind

(*undefined* | *create-object* | *destroy-object* | *alter-field* | *remove-key* | *manage*)

undefined [change kind not set]

0

create-object [new object added to the repository]

1

destroy-object [existing object removed from the repository]

2

alter-field [field was altered—remove old value, put new one]

3

remove-key [key removed from qualified field]

4

manage [management data have changed (named objects, ids, etc.)]

5

ch-body

(*create-body* | *other-body*)

create-body

class-name **;** *item-id* **;** *field-key* **;** *tx-id*

class-name [classhandler and affected item name]

v: (**class:***handler-name* | *handler-name*) **:** *item-name*

item-name [affected item name]

string

field-key [qualified field key]

(*item-id* | **v::***classhandler* | **-** ;)

item-id [transaction and sub-transaction identifier]

i : *id-prefix* # *id-index*

other-body

item-id ; *field-name* ; *new-value* ; *old-value* ; *field-key* ; *tx-id*

Transaction Record

transaction

t ; *tx-id* ; *tx-name* ; *tx-time* ; *tx-over* ; (*modifier*) ;

tx-id [transaction and sub-transaction identifier]

id-prefix # (*id-index*)

id-prefix [transaction part of identifier]

(**char**)5

id-index [sub-transaction part of identifier]

hex

tx-over [overarching transaction]

tx-id | **-**

tx-time [universal time of transaction]

UTC

tx-name [name of transaction type]

newUMLProject | **projectPreferences** | **newStoryBoard** | **newClassDiagram** | **newActivityDiagram** | **logicalStoryPattern** | **storyPatternWithThisObject** | **stop** | **createOrGotoMethodBody** | **editClass** | **editMethods** | **editAssoc** | **editActivity** | **editAssertion** | **editTransition** | **editLink** | **editCollabStat** | **inplace editing** | **include.includeClass** | **include.updateClass** | **editClassDiagram** | **globalEditAction** | **setTextComment** | **change size** | **drag** | **deleteDiagram globalDeleteAction** | **exportAndCompile**

modifier [source of transaction modification]

(*invalid* | *transient* | *local* | *server* | *application*)

invalid

0

transient

1

local

4

server

8

application

100

Appendix G

Mirador Source Code Metrics

Basic source code metrics for Mirador were collected over the course of development. The measurements were made with a shareware tool called Source Monitor. The measures from the last snapshot are given in Table 8. To Source Monitor, "Lines" is a count of the physical lines in the source files, and "Stmts" a count of the computational statements ending with a semicolon character, as well as the branching statements such as `if`, `for` and `while`.

Table 8: Mirador v0.84 Java source code statistics

Files	Lines	Stmts	Branch %	Calls	Cmnt %	Classes	Mthds /Class	Stmts /Mthd	Max Depth	Avg Depth
107	21,050	7,878	17.7	5,187	34.7	123	6.84	6.18	8	2.16