Comparing and merging UML models in IBM Rational Software Architect : Part 3
A deeper understanding of model merging

Level: Intermediate

Kim Letkeman (kletkema@ca.ibm.com), Development Lead, Modeling Compare Support, IBM Rational
02 Aug 2005

IBM Rational Software Architect (IRSA) is built on the Eclipse IDE and shares Eclipse's compare support workflows. IRSA UML models are built using the Eclipse Modeling Framework, so cannot be safely merged using the default Eclipse text compare support. This article, Part 3 in a series, discusses how you can manage the complexities involved when comparing and merging structured data like UML models.

Introduction

If you have read the first two articles in this series, you are by now familiar with the mechanics of comparing and merging models in IBM® Rational® Software Architect (IRSA) using Eclipse-style compare support features (see Resources, for the other parts in this series). But you may not yet have an appreciation for the complexities you need to manage in order to successfully compare and merge highly structured data like UML models. Relationships between elements within the model create dependencies that you cannot easily parse just by reading a list of physical changes between models.

Software Configuration Management (SCM) systems such as IBM® Rational® ClearCase allow parallel development without intervention by the modelers, but they force a merge session when parallel changes are detected. This in turn puts a great deal of pressure on you to understand two change sets and merge them in a way that retains the spirit of both without corrupting the model. Without sophisticated merge support, there are many ways that a model can be corrupted -- or data lost -- during a merge operation. One solution commonly practiced by modeling tools or teams is to store the logical model in fine-grained physical artifacts while practicing strong ownership at the physical level to avoid merge sessions. While this may appear to allow merge-free parallel modeling to occur, this method has its own issues.

The following introduction to parallel development and merging issues may be familiar if you have compared and merged structured data or models in previous generation tools. If you're new to modeling in team environments, these issues have made merging structured data very challenging. The rest of the article explores issues that can affect model integrity, and discusses the IRSA technologies that address them.

Fine-Grained Artifacts

Models that physically store elements as fine-grained artifacts can either create conflicts in the wrong context, or miss the conflicts entirely. Conflicts appear when the same elements are changed in parallel; however, with fine-grained artifacts, many changes manifest as artifact deletes and adds in the file system. The SCM system usually processes file system changes before it checks for logical content changes, so many changes will first be seen in the context of changes to a directory rather than in the model's context.

For example, a model might contain a package that contains a class, which itself contains attributes and operations. Storing this in a single physical artifact forces a merge in the model's context whenever parallel changes occur to that package. But storing each logical element as a separate physical artifact forces merges at the physical level, where there is no logical context. The SCM system will merge directories according to its view of what happened, and may not even notice conflicts.

An excellent example of how this can happen in Concurrent Version System (CVS) follows. A simple model stored as fine-grained artifacts might manifest in the file system as shown in code listing 1.
Now imagine that user A and user B check out the current version (Version 1) of this model. User A adds operation2 to class1 and user B deletes class1. So now you have two new versions of the model's file system representation as shown in code listings 2 and 3.

**Code Listing 2. User A's Model**

```plaintext
<<directory>>/mymodel
 mymodel.mdl
 <<directory>>/package1
 package1.pkg
 <<directory>>/class1
 class1.cls
 <<directory>>/attribute1
 attribute1.att
 <<directory>>/operation1
 operation1.att
 <<directory>>/operation2
 operation2.att
```

User A commits the additions first. Since this is the first of the parallel changes, everything goes smoothly. User B then commits the deletion of class1, which manifests as a delete of the entire file system hierarchy under and including `<<directory>>/class1`. CVS detects the second set of changes, but *does not flag the conflict!* It happily merges the changes, reinstating the parent directory hierarchy if necessary, as in this case.

When both users finally get synchronized, they end up with the model shown in Figure 1:

**Figure 1. Actual Result of Synchronizing these Parallel Changes in CVS**

Since CVS does not treat this case as a conflict, you end up with a hybrid of the changes that meets neither contributor's intent. Your model is somewhat corrupted, although not likely in a fatal manner. The reason may or may not be obvious -- CVS treated the changes as simple file system operations and performed an
intelligent merge of them in that context. The contributors actually made the changes in a modeling context, where a higher level of semantics normally applies, so the intent was quite different from the result.

The main point here is that conflicts are unavoidable if parallel development occurs at all. The best defense against nasty surprises -- like that shown in this example -- is to merge parallel changes in the context in which they were made. In this case, a simple conflict resolution would prevent model corruption, but that requires that you merge in the model's context. Since modern SCM systems work at the artifact level, fine-grained artifacts will inevitably have problems like that documented in this example.

This further implies that more context in an artifact improves the merge experience. Even if the SCM system in this case had detected the conflict at the **directory** level -- that is, adding content to a directory conflicts with deleting that directory -- the merge would still have had to be performed at the physical directory and artifact level, which is not the context in which the change was made.

There are numerous variations of this issue, but they all boil down to the fundamental advantage that larger artifacts have over smaller artifacts -- context during compare and merge operations. This is born out by empirical evidence gained from working with IRSA, CVS, and ClearCase.

**Change Atomicity**

Many gestures create many physical changes to the model, as element changes are made and relationships between elements are adjusted. Accepting some but not all of the changes created by a single gesture can cause the model to become semantically corrupt, no longer readable by its editor or other applications. The model can still be perfectly correct in both syntax and low-level (EMF) semantics, but its high-level (UML and notation) semantics are broken and often impossible to repair in a text editor.

An example of this would be a relationship (for instance, a line on a drawing) being retargeted from one element to another. In UML, a gesture for this might be to drag a generalization connector (obviously while editing in the context of a class diagram) from one class to a new class. This could easily happen when the specialized class wants to move up or down the inheritance tree.

In this case, the GUI will handle all the adjustments to the model -- a much better solution than simply deleting the old generalization and adding the new one, which would likely force an update on more than one diagram in the model. This gesture will generate a number of delete and add changes in the physical model, as semantic and notational data and references to source and target end points are changed.

*All of these raw deltas must be accepted or rejected at the same time,* or a serious inconsistency will appear in the model. Any inconsistency of this type might cause the model to fail to open later, effectively corrupting the model.

**Conflict Synchronization**

Resolving conflicting changes to opposite contributors is another source of potentially fatal model corruption. Related conflicting changes should be resolved to the same side in almost all cases.

One example might be when two modelers working in parallel retarget a generalization to different levels in the class hierarchy. As shown later in this article, this causes a number of related changes to both the notational (diagram) and the semantic (classes, packages) portions of the model. Since two contributors have changed a number of the same elements and attributes (for example, each has deleted the source class's target property and added a different target property), we now have the change atomicity issue (discussed in the previous section) interacting across two contributors. Obviously, this offers new opportunities during a merge session for you to accept parts of these changes from each side, thereby committing only part of each contributor's changes to the merged model and potentially corrupting the model.

A second variation could occur if a class diagram resides in a physical model that is separate from the main (shared packages) model. In addition to the interaction issues mentioned in the previous example, you would now have an additional problem of synchronizing the conflict resolution across multiple files. As a logical model is decomposed more and more finely, this synchronization issue grows in prominence. It can get really extreme in the fine-grained artifact case. Imagine a significant number of parallel changes...
across a couple of dozen physical artifacts. Each conflict situation pops up a visual merge that must be
resolved. As the individual artifact merge sessions are presented one by one -- for example while you are
performing a check-in or workspace synchronization -- you must try to remember the related resolutions
that went before in order to continue resolving all related conflicts to the same contributor. Failure to do so
will introduce inconsistencies between artifacts or between notation and semantic data.

Although it is probably obvious by now, this is another issue that supports coarse decomposition of
modeling artifacts.

**Clutter**

User interface clutter is a common problem with any merge tool. Examples include a text merge that
shows changed lines in multiple colors; an XML merge where changes in the ancestor and multiple
contributor artifacts have placeholders to mark insertion and deletion points for adds, deletes and moves;
and difference lists in previous generation model merge tools showing a flat list of all deltas without
sorting or hierarchical organization.

In all of these cases, you are confronted with a dazzling array of changes that you must sort out before you
can successfully merge the models. Handling each change separately can take many hours if there are
many weeks of competing changes. There can be hundreds of changes in a file of any significant scope.

The following examples describe specific issues that have plagued previous model merging tools.

- Say that you have a diagram containing thirty elements and you shift it slightly down and to the
  right to make room for a new element. This operation causes a change in the x and y coordinates for
each element, which generates sixty deltas. If these are not somehow isolated by diagram -- or even
  further by the multi-drag gesture that created them -- then massive clutter will obscure important
  changes.
- If you make related notational changes (for example, a change in fill color across the board) to all
diagrams, it will generate hundreds of deltas scattered across all the diagram contexts. Without
diagram-based compositing, these will again clutter up the merge session, making it difficult to see
important differences.
- Finally, if you add or change relationships on several diagrams, it will create many raw deltas, again
  scattered all over multiple diagrams. If there is no organization by diagram, and then again by
  gesture, these important deltas will obscure each other and be very difficult to accept or reject
during a merge. Conflicts in this scenario are a nightmare, because relationships between raw deltas
  become visible.

**Summary**

As you can see, there are many issues with merging structured data. From this point, IRSA technologies
and practices for mitigating or eliminating these issues will be discussed.

**Model Partitioning**

In some modeling tools, it is advisable (or at least advised) to partition models in order to minimize
parallel development at the artifact level. Theoretically, this reduces the likelihood of non-trivial merges
(those that require user intervention because of conflicting changes).

With IRSA, it is not necessary to partition models just to try to avoid merges. With real-world models, this
strategy cannot fully protect you from merging anyway (as there are always shared artifacts, and they will
often have parallel changes). You are instead encouraged to practice strong logical decomposition and
ownership using packages or artifacts. The goal is to keep the model as physically intact as possible, with
only the shared packages under a lot of parallel development pressure in a separate artifact.

Merging with IRSA is much faster than with previous solutions (a speed increase of 10^4 in the delta
generator and conflict analyzer), so there is no incentive to shrink the files just to reduce the amount of
processing time during the merge, even for very large models. Also, reducing the number of physical
artifacts in a model will speed up processing in the SCM system during synchronization, which can save a lot of time during workspace synchronization. At the extremes you can have only one model artifact; or you can have hundreds or even thousands of controlled units. Imagine the difference in time required for repository operations.

There are definitely circumstances under which model partitioning will be helpful or even necessary. The need for partitioning will manifest as a performance issue (that is, a lack of resources to handle the model during normal operations). As models grow, they:

- Require more space on disk
- Use more memory while you are editing
- Require more bandwidth when loading from disk or transferring over the network, including VPN connections from remote locations

More memory use means more paging, which brings with it a lot of pauses in the application. These are signs that the model may be getting a bit larger than your computing resources can handle.

So let's say that you've decided to partition your models to speed up your day-to-day work and reduce the memory footprint, perhaps extending the life of older or under-resourced PCs. There is a right way and a wrong way to approach the task, and the difference between them is essentially the difference between a working solution and an utter failure. There is very little grey area here.

Wrong Way

Your first instinct might be to chop your large model up into separate packages and diagrams based on what people happen to be working on at the time, creating dozens of small pieces. If this is done without a lot of thought and organization up front, then many cross-model references will be created. The effect of this is that most artifacts inter-relate with many others, and large portions of this mesh will be loaded into memory whenever you open any of the pieces. The merge footprint will be much smaller, but the context will be very limited since the pieces are small and not logically cohesive.

In this case, you've not only made no reduction in the effective memory footprint, you've actually made things more cumbersome. Synchronization has to process more files, and merges are more challenging because each merge has only a small subset of the necessary context.

Spaghetti models are as bad as spaghetti code: worse actually, since the physical partitioning is bound to confuse users of the model because they will automatically attach significance to the partition boundaries. All in all, this method is actually detrimental.

Right Way

On the other hand, you may choose to spend a lot of time up front logically partitioning your model into separate cohesive packages that have minimal overlap and inter-dependency. For example, your use case model would not overlap with your deployment model. Neither would overlap with your dynamic behavioral models (interactions, state machines). And so on. You can visualize these relationships between packages in the overall logical model as a wheel -- with the shared packages at the hub and your logically cohesive packages on the spokes as shown in Figure 2.

Figure 2. Partitioned model with reduced memory footprint
Logical partitioning enables strong ownership of specific model areas -- UML diagrams and packages -- and has the effect of minimizing conflicting changes when modelers work in parallel. These packages may be separated physically at any time. But since strong logical ownership is all that is really necessary to ensure trivial merge sessions, there is no immediate need for physical partitioning unless resource limitations are driving.

Theoretically, a team with ten members could always have a parallel development stack ten deep. As each person checks in a modified version of the artifact, it spawns a merge session for the current change set and all of the previously accepted change sets.

This may sound a bit cumbersome, but in fact it is an effective work flow when used with CVS and ClearCase. These SCM systems have excellent integrations with IRSA (discussed in some depth in future articles in this series), and are flawless when detecting parallel modifications and forcing merges. Note that the frequency and scope of merges can be substantially reduced by regularly updating your workspace to the latest baseline in the repository. That way, you always start from a version that already contains most of the recent changes.

With strong logical ownership, merges are generally of a trivial nature -- performed automatically in ClearCase or requiring minimal user intervention in CVS. And when your changes affect the same elements, IRSA is adept at isolating the conflicts (discussed in more depth later in this article), and at presenting the choices clearly for user action.

Once this state of logical partitioning is achieved, then physical model partitioning can proceed if necessary.

For a very detailed discussion of model structure and partitioning guidelines, see the Developer Works white paper entitled Model Structure Guidelines for Rational Software Modeler and Rational Software Architect by William T. Smith of IBM.

**UML versus EMF**

IRSA is implemented in the Eclipse integrated development environment (IDE) platform. IRSA Unified Modeling Language (UML) models are likewise implemented in the Eclipse Modeling Framework (EMF), an open source modeling language that provides a significant tool set for developing and implementing meta-models and the applications to use them. UML model compare support is implemented using a generic EMF differencing engine, which dramatically affects the way that you and the software perceive model differences.
The UML is a logical language that represents the elements of software systems from several different perspectives. Each perspective is typically modeled in a diagram, with some examples including class relationships, operational sequences, components, state machines, and so on. The logical structure of the model must be represented physically -- in memory and on disk -- and EMF provides the API by which this is done in IRSA. The logical-to-physical mapping is relatively straightforward and is shown in Figure 3.

Figure 3. Logical UML model represented as physical EMF model

As an example of this mapping from the logical UML model to the physical EMF model, look at a relationship between two classes: say a generalization from class1 to class2. You would expect to see a single difference between the before and after models, looking something like:

```
Added a generalization relationship -- Class2 generalizes Class1
```

However, adding a generalization relationship on a diagram actually requires five physical changes to the model, although only four are typically visible:

- A generalization element is added to the more specialized of the two classes. This creates a single semantic relationship in the model, allowing the next three deltas to be created once for every diagram on which these two end points are subsequently dragged.
- An edge to represent the generalization connector is added to the diagram.
- Two changes occur when the node at each endpoint of the generalization adds a reference to the edge. Each reference represents either the source or target end of the generalization.
- A reference to a target class is added to the generalization element. This reference is often hidden because it is grouped by containment with the add or delete of the generalization itself. Only a change of the target endpoint can expose this reference as a delta, which you will see further on.

These model differences at the physical level are closely related to each other. If the tool leaves them scattered around the structural viewer, it is very difficult to put it all together and easy to get them into mismatched states. This would be the equivalent of accepting only part of a user gesture, and would result in significant confusion and potential model corruption.
Although IBM Rational tools have implemented technology like composite deltas to help group these raw deltas by user gestures and by diagram, it will help you to go through some examples of how these low-level (raw) deltas are presented, and of how their formats and other visual cues are used to help explain the meaning of the raw deltas that you see. After these examples, the article will discuss model corruption and the features that exist to prevent it when you compare and merge models. I hope that understanding delta formats and corruption prevention will improve your confidence when you begin comparing and merging models.

**Terminology**

- **Delta** or **Difference**: Any physical difference between two models. Delta is typically used in the context of EMF because the Delta Generator uses this term. Difference is used to refer generically to a difference between two models. However, the two have become interchangeable so please read them as exactly the same thing.

- **Dependant delta**: A delta (difference) that must be rejected when another is rejected. An example is an add delta that adds a package into which a move delta will insert an already existing class. If the add does not take place, then the new parent package will not exist and the move cannot take place. Therefore, the move operation is a dependant of the add operation. If the add delta is rejected by the user, then the model integrity protection feature will force the move delta to be rejected automatically as well, preserving the moved data. Also see prerequisite delta.

- **Edge**: A line on a diagram that represents a relationship between two nodes (for example, two classes.) An edge can be an association, a generalization, a dependency and so on. Think of an edge as a view onto a relationship in the model, as shown in Figure 4. Also see node.

**Figure 4. Nodes and Edges**

- **Model integrity protection**: A mechanism by which prerequisites and dependants are automatically processed in order to avoid data loss or corruption. You could also include atomic composite delta processing as a factor in model integrity protection.

- **Node**: A figure on a diagram that represents a semantic element (for instance, a class, an interface, and so on). Think of a node as a view of the semantic element in the model. Also see Edge.

- **Notational element**: A visible element (represented by a shape) that represents a semantic UML element in the context of a single diagram. See node or edge. Any number of notational elements can exist to represent the same semantic element on different diagrams. A notational element tends to have properties that affect the visual representation of the semantic element; for example, x and y coordinates and height and width are notational properties. Also see semantic element.

- **Prerequisite Delta**: A delta (difference) that must be accepted when another is accepted. An example is an add delta that adds a package into which a move delta will insert an already existing class. If the add does not take place, then the new parent package will not exist and the move cannot take place. Therefore, the add operation is a prerequisite of the move operation. If the move delta is accepted by the user, then the model integrity protection feature will force the add delta to be accepted automatically first. Also see dependant delta.

- **Semantic Element**: An element that represents a physical entity or a relationship such as a class,
interface, generalization, association, actor, and so on. These elements have semantic meaning and are attached to models and packages. They are represented within diagrams by equivalent notational elements (nodes and edges.) Also see notational element.

UML and EMF Differences

IRSA provides a full suite of UML editing, visualization, transformation, and patterns tooling, allowing you to create large and complex models and software systems. The UML is represented in the model using semantic and notational data that itself is physically defined and implemented in terms of meta-models defined in EMF. When a UML model is serialized to disk and de-serialized back to memory, all operations are performed using the EMF API. As mentioned in the introduction, a single change at the UML level -- for example, adding a generalization relationship between two classes -- is implemented as multiple changes at the EMF level.

Since the delta generation engine is implemented at the EMF level, the EMF deltas must be related back to UML tool operations and gestures for user consumption. An understanding of how UML changes are represented in EMF can therefore be of significant assistance in developing a deeper understanding of how to compare and merge UML models using IRSA.

Following are several examples highlighting this UML-to-EMF delta mapping. Concepts are introduced and explained as needed to enrich your understanding of the compare and merge process.

Example 1: Add a Generalization to a Diagram

This often-used example illustrates the differences found at the EMF level when you perform a single add generalization gesture at the UML level. After adding the generalization relationship to a diagram, you will compare the new model with the previously saved model using Compare With Local History. A generalization is actually a semantic relationship between two classifiers, and will therefore appear on any diagram in which the two classifiers themselves appear unless the line is explicitly deleted from the diagram.

Adding the generalization
Continuing with the example at the end of Part 2 of this series of articles, you start with two classes as shown in Figure 5.

Figure 5. Before adding the generalization

![Image of Figure 5](hw1.emx)

In figure 6, you select Generalization from the palette. You then drag the end point from Class1 to Class2 and you see the result shown in Figure 7.

Figure 6. Selecting generalization from the palette

![Image of Figure 6](hw1::ClassesDiag.png)
Figure 7. After adding the generalization

![Diagram showing Class1 and Class2 with a generalization arrow]

This required a single UML gesture, but resulted in a series of changes to the underlying EMF instance document.

**Deltas**

After comparing the two versions using the local history facility, you can now see the raw EMF deltas and their composite difference groups, as shown in Figure 8.

Figure 8. Four EMF deltas

The four EMF deltas from top to bottom are:

1. A reference to a generalization added to Class1's source edges collection. The (Class1)(Class2) notation is used for associations whenever the ends can be determined and indicate in this case that the arrow points from Class1 to Class2. The existence of this notation indicates that the generalization denoted in this delta is a *view* on the element, not the semantic element itself. This notation further indicates that Class1 is a more specialized version of Class2 -- in other words, that Class2 generalizes Class1. The [reference] qualifier indicates that this delta is for an added *reference* (connection) to an element, not an added element.

2. A similar reference to the same generalization view added to Class2's *target edges* collection. This indicates again that Class2 is the more general class in this relationship (where the arrow ends).

3. The edge (a line or arrow) on the diagram, representing the generalization visually. This is the *generalization view* that is referenced by the first two deltas. The edge is added to the diagram's edges collection. An edge is a notational meta-model entity representing a connector, and always points to a semantic element that provides meaning for the association.

4. The semantic generalization element, which is owned (contained) by Class1's *Generalization* attribute. This denotes a semantic relationship between the two classes at the UML level. It has nothing specifically to do with any one diagram. On the other hand, the previous three deltas are notational and represent changes to the visual representation of the generalization in *one specific* diagram. The final delta is the only delta that represents the actual semantic meaning of a generalization. It is grouped in this case inside the diagram and relationship composites because diagrams and relationships have a strong affinity for any semantic data that is added simultaneously with its representation on a diagram. If you remember, I earlier referred to a fifth change to the model when adding a generalization -- that being the generalization's target parameter. This change is actually contained within the semantic generalization element, so its addition is not visible to you.

**Recap of delta notation**
It will be useful to recap the delta formats used by IRSA UML compare support. Figure 9 shows the format of the five delta types, and illustrates the three main fields in each one.

Figure 9. The five delta types

Each EMF delta contains these fields:

- **Delta type**: A change to a specified object or attribute in the model. Delta types are *Added*, *Changed*, *Deleted*, *Moved*, and *Reordered*.
- **Affected object or attribute**: The name and type of an object, optionally including an attribute name for change deltas. The format is *name <type>.attribute name*. The name and attribute name are optional. For comments and other unnamed text fields, a pseudo-name is crafted from the first part of the contents in double quotes. The type uses angle notation: *<type>*. Optional additional qualification of the name can include the following:
  - **Reference qualifier**: Indicates that the change (addition, deletion, and so on) operated on a reference to another element and not on a contained element. Notation is *[reference]*.
  - **Stereotype qualifier**: A list of stereotypes and keywords associated with the affected object. Notation is *<<kw1,kw2,kw3...,st1,st2,st3...>>*.
  - **Edge qualifier**: One or two endpoint designators specific to an edge. Notation is *(from endpoint)(to endpoint)*.
- **Change description**: Contains parent or target location for add, delete, and reorder deltas. Contains source and target locations for moves. Contains *before* and *after* attribute values for changes.

**Semantic generalization**

In our example, the fourth delta denotes the addition of the semantic generalization element. The generalization is an unnamed element (no name shown on the diagram or in the model explorer) so the element is shown in the delta summary by its type in angle brackets *<Generalization>*. The generalization element is a child of (or owned by) the class *Class1*. This parent-child relationship is read as *Class1 is generalized by Class2*. In the model explorer, the element is denoted as a generalization only by the generalization icon; the class that generalizes the owner is named in ellipsis notation denoting the more general class, as shown in Figure 10.
Figure 10. Semantic generalization element denoting that Class2 generalizes Class1

The highlighted delta is shown in Figure 11.

Figure 11. Highlighted addition of semantic generalization element

In figure 12, you can see that the change is structural in nature, as added semantic element data should be, and is shown in the Model Explorer.

Figure 12. Highlighted semantic generalization element

With this semantic relationship in place, adding Class1 and Class2 onto any diagram will now automatically show this relationship if the diagram is able. For example, if you add a second class diagram to the model (as shown in Figure 13), you can then drag the two existing classes onto the diagram and the class nodes and generalization edge are automatically added for you (as shown in Figure 14).

Figure 13. Added second class diagram

Figure 14. Second class diagram

Each diagram has its own view onto the relationship. It is, in fact, possible to delete the view from one diagram while retaining it on another. This is probably a bad idea if the relationship makes sense on that diagram. On a diagram representing a different context (for example, a sequence diagram) the relationship would not appear even though the classes might.

Viewing semantic data with nodes and edges

To review, when a semantic element is represented on a diagram, it is viewed through a notational element (node or edge). A class is viewed through a node and a relationship through an edge. When an edge joins two nodes on a diagram, the connection points are implemented as source and target references to the edge. In the previous example, the nodes already existed on the diagram. When you added the relationship and compared the model with the previous version, you saw the additions of these references explicitly as
individual deltas (remember the source and target edge collections.)

In the second class diagram that you just created, the relationship appeared automatically when you dragged the two classes onto the diagram because it already existed as a semantic element under Class1. You saw only the addition of the two nodes pointing at the classes and the edge pointing at the generalization, as shown in Figure 15. The reference connections are also created, but they are contained in the new class nodes and thus are not separately called out as deltas. Remember that additions of elements in parent and child configurations (trees or branches) are represented as a single add delta at the root.

**Figure 15. Deltas created by dragging two classes onto second class diagram**

![Diagram](image)

The first two deltas represent the creation of the nodes for Class1 and Class2 on the diagram. The third delta represents the edge for the semantic generalization element. It is owned by Class1 and pointing to Class2, and you see that reflected in the delta. You will see either of these patterns of deltas every time you add a relationship to a diagram. You will see similar patterns when you delete a relationship from a diagram.

**Example 2: Change the Generalizing Class**

So what do you see when you simply change the target endpoint of a generalization?

To find out, add a third class to the diagram and save the model. Then change the generalizing class from Class2 to Class3 (by dragging the arrow endpoint as shown in Figure 16) and see what sort of deltas you get (figure 17.)

**Figure 16. Changed the generalizing class**

![Diagram](image)

**Figure 17. Deltas from changing the generalizing class**

![Diagram](image)

Essentially, you've deleted Class2 and added Class3 as your generalizing class. In EMF, many attribute changes use a delete-add mechanism instead of a change mechanism. These deltas are grouped together, so the delete and add deltas are performed at the same time, which is equivalent to a change delta at the UML level. The first, third, fourth, and sixth deltas are quite familiar from previous examples.

But the second and fifth deltas are new. This is the first time that you've made a change that exposes the name of the attribute containing a reference to the other end of the relationship. The semantic generalization element has a target attribute that stores a reference to the other end (Class2). You changed
that using the delete-add method from a reference to Class2 to a reference to Class3. This is the hidden-containment-delta issue that was mentioned earlier in the article. Although the context-sensitivity of additions can be disconcerting -- that is, an added element will not show up as a separate delta if its parent was also added -- it is very necessary to reduce clutter and preserve the performance of the merge application.

Atomicity
So if you model these changes as deletes and adds, can't you create problems during a merge scenario by accepting the add deltas and rejecting the delete deltas? This would seem to leave both the old and the new elements and references in the model, which implies that there would be two generalizations, a serious violation of UML model semantics. Since there is only one Generalization attribute, this would likely be a fatal model corruption scenario.

The solution to this particular class of problems is the atomic composite delta. The relationship composite delta shown in blue highlighting in Figure 17 has a two-triangle delta group icon with the addition of the atomic grouping symbol (four dots at the corners.) Accepting or rejecting any of the deltas in this group will automatically perform the operation on the whole group. All relationship gestures are grouped atomically in this way. This protects the application from having to handle partial changes or outright model corruption caused by a merge in which the user tried to pick and choose pieces of gestures.

Example 3: Multi-Drag

Not all atomic groups exist to protect against corruption.

One class of atomic groups handles the familiar drag and mutli-drag gestures. Say that you select all the objects on the diagram and drag them a short distance at an angle, creating some amount of delta in the x and y directions. Since you have three classes and a relationship visible on the diagram, you might expect to see eight deltas: one for each x and each y. On a diagram of any size, this would create a lot of clutter, and would be almost impossible to process during a compare or merge without some sort of grouping mechanism.

IRSA creates a multi-level drag composite to isolate the gesture. Figure 18 shows how this looks in the compare session with the ghost image of the other contributor turned on -- the before image is ghosted in this case -- to show the drag. Remember that the button to show the other contributor is located in the top right-hand corner of a contributor's pane.

Figure 18. All objects dragged

![Image of all objects dragged](image)

The deltas in their hierarchical form are shown in Figure 19.

Figure 19. These deltas are a drag

![Image of deltas](image)
You might be surprised that only the nodes generate deltas for a drag gesture. This actually makes perfect sense when you consider the fact that the edge is anchored to its end points (as you saw in several delta scenarios earlier.) The edge moves and stretches or shrinks along with its end points so it needs no notational bounds data of its own. The six x and y deltas are grouped first by the dragged object, and those dragged objects are then grouped by their common delta x and delta y amounts in a multi-drag gesture group. A multi-drag group is created only with objects that are dragged exactly the same distance and direction. Single drag groups can exist independently, and can even contain only one delta if the object was dragged along the x or y axis.

Now, back to atomicity. You can see that all of these gestures are atomic by the four little dots on their group delta icons. Compare support is not intended as a funky editor, so it makes gestures atomic as often as possible to avoid having users perform partial operations that were impossible in the original editor or were not made by one of the two contributors. Accepting or rejecting any x or y delta therefore accepts or rejects them both together. The element jumps back and forth between its source and target locations without ever visiting the other possible locations denoted by \((x, !y)\) or \((!x, y)\).

Note that it is possible, if unlikely, that a multi-drag composite can be accidentally created by dragging individual elements along the x and then the y axes or along any angle a number of times, and arriving by chance at the same delta x and delta y. IRSA always treats this case as a multi-drag.

Atomic hierarchies
Atomic delta groups implement atomicity for everything underneath them, down to the lowest level. Thus, accepting or rejecting any of the x or y deltas anywhere in a multi-drag delta will perform the same operation on every delta at every level of the tree underneath the multi-drag parent. This takes some getting used to, because a lot of deltas suddenly change state when a single delta is accepted or rejected. The Model integrity protection section will try to explain exactly how complex this can really get.

Example 4: Sequence, Activity, and State Machine diagrams

Another atomic group exists to protect the integrity of the dynamic diagrams.

Three kinds of diagrams are themselves atomic. They are Sequence diagrams, Activity diagrams, and State Machine diagrams. In fact, for these diagrams, their entire Interaction, Activity, and State containers are atomic. This is because these diagrams are reflections of contained semantic data and have behaviors in them that are not purely notational. Our goal is to enable the more complex merging of these diagrams in a future release.

Parallel changes to these diagrams automatically create atomic composite deltas which may or may not conflict in their element changes. Since it is possible to accept both sets of changes (with unpredictable results) IRSA creates a composite delta conflict, summarized in the structured differences viewer as \(\text{Conflicting difference groups}\) (shown in Figure 20). What this means is that parallel changes to these three diagrams will conflict wholly, so only one or the other change sets will be preserved. This is colloquially referred to as a pick A or B solution. There is no better argument for strong logical ownership of these specific diagrams. Hitting a merge situation is guaranteed to lose one or the other contributor's changes. This pick A or B behavior is specific to individual containers. You can accept user A's sequence diagram and user B's state machine in conflict situations that affect both diagrams. Diagrams never affect each other in this situation.

Figure 20. State machine difference group conflict
Conflict Analysis

In addition to atomicity as a model corruption prevention mechanism, there is a conflict analyzer searching for dangerous or incompatible delta combinations. For example, look at a classic example of two conflicting gestures that result in immediate corruption or data loss to the model. This is the circular move scenario, where two packages become the parents of each other in two contributors. The beginning of this scenario is illustrated in Figure 21.

Figure 21. Two packages

Now create two parallel copies of this base model; in the first make package1 the child of package2 (Figure 22) and in the other do the opposite (Figure 23).

Figure 22. Package 1 is package 2's parent

Figure 23. Second contributor, package 2 is package 1's parent

Now imagine that you accept both changes and create a containment circle. Depending on the underlying parent-child implementation, one of two things can happen when the model is traversed from the top:

1. The application encounters one of the two packages in the circle. It then traverses the child collection, encountering the other package. It traverses that package's child collection, encountering the first package again. This goes on for a while, pegging your CPU at 100% until you eventually kill the application.
2. The application never sees either of the packages, because they can have only one parent and each package has the other as its parent, with both being disconnected from the model's hierarchy.

The conflict analyzer prevents this by looking for any combination of moves that lead to a circle. Sounds simple, but isn't. It must handle any depth of separation between the moved objects and it must run fast (no quadratic algorithms, please).

This is but one example of the conflict strategies that are run each time a compare or merge session is started. Some conflicts exist simply because accepting both deltas would be impossible (for example, a
change to an element's name on one side and a delete of the element on the other.) But some are there to prevent fatal corruption of the model.

This is one reason why text editors make such poor structured data editors and merge solutions. There is no context by which such decisions can be made, much less made automatically without user intervention. So if you are ever tempted to edit your models in text form, be very careful and keep a backup. Better yet, grab a coffee and let the urge pass.

Model Integrity Protection

IRSA has one final corruption prevention feature, and it's the big one: model integrity protection. This feature:

- **Prevents elements from being lost**: For example, when an element is moved into a newly added package. If you accept the move delta but reject the add delta, you inadvertently throw away the moved element.
- **Prevents references from being broken**: For example, when a class is added to a diagram and the class node references the semantic class element. If you accept the added class node but reject the added semantic class element, the class node has a broken reference. This type of scenario can cause fatal model corruption without model integrity protection.
- **Prevents deltas in all contributors from getting out of sync with each other**: For example, when you have a number of related atomic groups with conflicts and mutual prerequisites or dependants. These relationships between deltas and conflicts in the ancestor and two contributors can become staggeringly complex, but the cascade feature in model integrity protection handles this without problems. The effect is of an organic process where many deltas change state as the result of a single delta acceptance or rejection.

Prerequisites and Dependents

All of these scenarios rely on the processing of prerequisites and dependants. A prerequisite is a delta that must be accepted in order for another delta to be successfully accepted. A dependant is the same relationship in reverse: a delta that must be rejected when another delta has been rejected. These are invoked automatically when accept and reject commands are processed, and they also come into play when changes must be reversed by an undo command.

To illustrate the need for a prerequisite, use the previously-mentioned example. A class was added to the diagram, creating a semantic class element in the model and a notational class view element in the diagram. The class view references the class element directly. If the class element does not appear in the model, then the added notational element will have to display a broken reference symbol if it can. In some applications, this scenario can result in a failure to open the model; hence model integrity protection must be applied automatically.

Imagine now that the deltas are all sitting there in an unresolved state. The user sees the added class view and accepts the add delta. If the added semantic class delta is never accepted, you now have a broken reference. But references are specifically maintained by model integrity protection so that the prerequisite delta is automatically found and accepted before completing the acceptance of this delta. This form of prerequisite is known as referenced element must exist.

Another form of prerequisite is all parents must exist. This comes into play when a hierarchy of packages is added to a model and an element (say, a class) is moved from a pre-existing package to one of the new packages in the added hierarchy. This prerequisite forces the add delta to be applied before the move is applied, because the target package must be in the model for all of its parents to exist.

Figure 24 illustrates this mechanism.

**Figure 24. Add delta is prerequisite of move delta**
Note that model integrity protection using prerequisites and dependants works in one direction. For example, a class node references a semantic class element, but the semantic element does not know about the node. So accepting an added class node will force the acceptance of the added semantic element to which it points in order to prevent a broken reference. But accepting the added semantic element has no prerequisites, since there is no reference in the other direction. In other words, you can have only the semantic element, but you cannot have only the class node (it must point to something in the model.) Although this example seems pretty intuitive, things can quickly become much more complicated.

**The Cascade**

Imagine that a single delta is accepted in the left contributor of a three-way merge session. It happens to have an atomic composite delta somewhere up its parent hierarchy, which causes a few dozen other deltas to be accepted as well. Furthermore, each of these may have one or more prerequisite deltas that must be accepted. Each of these prerequisites, in addition, may be in other atomic composites, and so on until there is nothing left to process on the left side.

Next you find that some of the accepted deltas have conflicts in the other (right) contributor. For each of these conflicts, jump across to the other side and reject the conflicting delta. Now, if the conflicting delta was already rejected, then nothing further happens. But if it was not, then you change the delta's state. After the rejection is applied, you have to reject all of its dependant deltas. For each delta that actually changed state, you have to find the highest containing atomic composite and perform the rejection of all contained deltas. And for those, you have to perform the dependant rejection as well, and so on until you are done on the right side.

Once you've done everything on the right side, you have to find all the conflicting deltas for those deltas that changed state. For all conflicts that have not already been processed once, you have to leap back to the left side to continue in the same vein. you leap back and forth until you run out of atomic deltas, prerequisites and dependants, and conflicts.

This is the cascade algorithm (simple version) and you only have to know that it works. But when you accept or reject deltas, be aware that you may see a lot of deltas changing state. This seemingly organic behavior is necessary to preserve model integrity.

---

**Final Word**

Merging structured UML data has in the past been a difficult and sometimes risky process. This article discussed the approach that IRSA takes to compare and merge UML models, and to prevent model corruption and data loss. Model and referential integrity are protected with a number of features, including atomic composite deltas, model integrity protection (prerequisite and dependant processing), conflict analysis, and the cascade algorithm. All of these -- working together with the powerful grouping of deltas in diagrams -- make for a relatively painless merge experience, even when there are contributors with conflicting changes.

Future articles in this series will explore parallel development in CVS and ClearCase team environments; and parallel development in the presence of custom UML profiles.
Resources

- Part 1 in this series, Comparing models with local history (developerWorks, July 2005).
- Part 2 in this series, Merging models using "compare with each other" (developerWorks, July 2005).
- Part 4 in this series, Parallel model development with CVS (developerWorks, August 2005).
- Part 5 in this series, Model management with IBM Rational ClearCase and IBM Rational Software Architect Version 7 and later (developerWorks, July 2007).
- Part 6 in this series, Parallel model development with custom profiles (developerWorks, August 2005).
- Part 7 in this series, Ad-hoc modeling – Fusing two models with diagrams (developerWorks, March 2007).
- The article Introducing IBM Rational Software Architect: Improved usability makes software development easier (developerWorks, February 2005) is a basic introduction to the Rational Software Architect product.
- Get the evaluation version of Rational Software Architect from the Trials and betas page.
- For technical resources about Rational's products, visit the developerWorks Rational content area. You'll find technical documentation, how-to articles, education, downloads, product information, and more. For specific information about Rational Software Architect, visit the RSA technical resources page.
- For details and more information about the Eclipse 3.0 platform, visit the Eclipse home page.

About the author

Kim joined IBM in 2003 with 24 years in large financial and telecommunications systems development. His responsibilities with the Rational Modeling Tools team include UML and EMF Compare Support, Architectural Discovery for Java and UML, Traceability, and Test Automation.