In Proceedings of the 2005 ACM/IEEE International Conference on Automated Software Engineering, Long Beach, CA, Nov 2005

A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design

Akhil Mehra
Dept. Computer Science
University of Auckland
Auckland, New Zealand
+64-9-3737-599
akhilmehra@gmail.com

John Grundy
Dept. Electrical and Computer Eng.
University of Auckland
Auckland, New Zealand
+64-9-3737-599 ext 88761
john-g@cs.auckland.ac.nz

John Hosking
Dept. Computer Science
University of Auckland
Auckland, New Zealand
+64-9-3737-599
john@cs.auckland.ac.nz

ABSTRACT
Differentiation tools enable team members to compare two or more text files, e.g. code or documentation, after change. Although a number of general-purpose differentiation tools exist for comparing text documents very few tools exist for comparing diagrams. We describe a new approach for realising visual differentiation in CASE tools via a set of plug-in components. We have added diagram version control, visual differentiation and merging support as component-based plug-ins to the Pounamu meta-CASE tool. The approach is generic across a wide variety of diagram types and has also been deployed with an Eclipse diagramming plug-in. We describe our approach's architecture, key design and implementation issues, illustrate feasibility of our approach via implementation of it as plug-in components and evaluate its effectiveness.

Categories and Subject Descriptors
D.2.2 **[[Software Engineering]]** Design Tools and Techniques – *CASE tools*

H.5.3 **[Information Systems]** Group and Organization Interfaces - *asynchronous interaction*

D.2.7 **[Software Engineering]** Distribution, Maintenance, and Enhancement – *version control*

General Terms
Design, Human Factors

Keywords
visual differencing, merging, version control, CASE tools.

1. INTRODUCTION
Efficient management of software artefacts is a major task of any project involving more than one person. An important goal is maintaining versions of software artefacts as they evolve, preventing people from accidentally overwriting each other's work, and allowing tracking of changes made to those artefacts over time [1], [3], [21].

A related and important task is support for version comparison and merging [1], [21]. Although configuration management tools provide good support for versioning, tool support is also needed to identify differences between versions of an artefact. The Unix tool diff [11] is one such popular tool for comparing two text files. Diff compares files and indicates a set of additions and deletions. Many version control tools provide functions similar to diff to identify changes between versions of text documents. Initially diff tools were hard to use as users needed to manually navigate to lines where changes were made. This led to creation of visual differentiation tools, to improve usability and highlight changes within IDEs. Many of these are generic working across many text-based document types [10], [15]. Many IDEs incorporate such differentiation facilities, often using a diff-style tool as a component-based plug-in to do the comparison with results presented visually in the IDE, as in the Eclipse version tree plug-in [2]. Merge support in an IDE uses differences detected to apply changes made in one version of a document to another.

Although good, generic support is available for differentiating and merging text documents, limited support is currently available for differentiating graphical objects such as UML design diagrams and software architectures [16], [22]. Providing visual differentiation in CASE tools is important to enhance a team's efficiency and effectiveness when collaborating asynchronously using diagrams to represent information [20]. Existing diagram differentiation tools are usually limited to a single diagram type and hard-coded into the CASE tool. A diff-style algorithm doesn't work for two (or three) dimensional diagrams as the isolation of the diagram into "islands of difference" is very difficult. Tool developers use diagram type-centric techniques that need recoding for different diagram types.

We have added generic visual differentiation facilities to Pounamu [23], a meta-CASE tool which allows a user to specify and generate multi-view visual design tools. Pounamu diagrams are stored in XML format so visual differentiation facilities were added by differencing this XML format. Differences are then translated into editing events which are presented to users by using appropriate highlighting to emphasize differences. Users also have the ability accept/reject changes made thus enabling partial merging of diagrams.

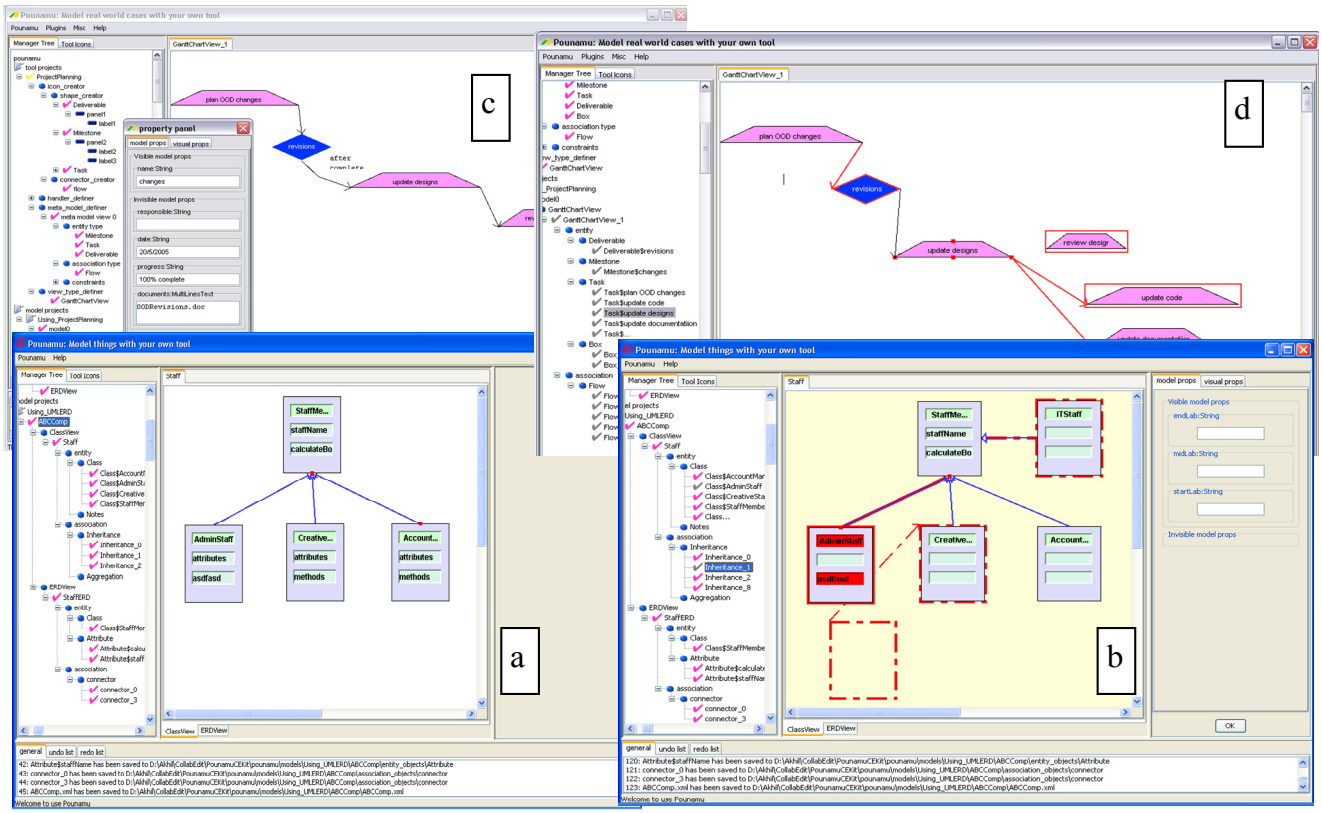

Figure 1. Examples of editing tools in Pounamu.

Syntactic and some semantic conflicts are detected and presented. Use of our technique on a range of diagram types has proven successful, and usability evaluations demonstrate its effectiveness.

We first present motivation for our work and related research. We then present our approach for providing versioning, visual differentiation and merging support in Pounamu and illustrate user interaction with our plug-ins. An Eclipse graphical editor plug-in for Pounamu-specified diagrams using the same differencing and merging plug-ins is also shown. We then describe key design and implementation decisions, evaluate our approach, and identify directions for further research.

2. MOTIVATION

Consider two users modifying the design of a system. These developers may work together to review and modify the design, but at times may be working independently and need to make design modifications. Figure 1 (a) shows part of a UML design for a software system that may be under modification by user "John". John may add classes, attributes, operations and relationships. He may modify existing classes and relationships. He may also reposition diagram elements, change relationship role names and arities, and possibly annotate the design diagram with notes or other supported annotations. User "Mark" will want to be isolated from these changes for a time, requiring versioning of design diagrams. Eventually Mark will need to compare the version of the design he has to that of John's via a diagram differencing facility, highlighting changes as shown in Figure 1 (b). Figure 1 (c) and (d) show other examples of diagram difference presentation, in this case a Gantt chart. Mark may want to merge some or all of John's changes into his version, which may include some of Mark's own independent modifications. Other kinds of diagrams John and Mark are asynchronously editing will also need to be versioned and differentiated and merged.

The diagrams in these examples are implemented using the Pounamu meta-tool [23]. Pounamu is a meta-tool for building domain-specific visual language diagramming tools. Pounamu also provides a framework for realizing and using the specified tools. Using Pounamu a user can rapidly specify visual notational elements, underlying tool information model requirements, visual editors, the relationship between notational and model elements, and behavioral components [23]. Tools are generated dynamically and can be used for modeling immediately.

Efficient asynchronous collaboration amongst groups of people working together may be facilitated using versioning and differentiation tools. We were motivated to improve asynchronous collaboration in Pounamu by providing users the ability to version Pounamu model projects via a CVS repository [9]. The ability to version Pounamu model projects led immediately to the requirement for generic visual differentiation and merging tools for diagram versions.. Such a tool should enable users to visually compare their current work with prior versions or other user's versions of the same diagram. It should also provide users with information about addition, deletion, movement or property changes in shapes or associations and users must be able to specify changes they wish to keep and changes they wish to discard thus enabling them to merge their current diagram with any prior version. Syntactic conflicts should also be presented to users for resolution, and any semantic conflicts

introduced by a merge should be highlighted. As Pounamu supports building a huge range of domain-specific diagramming tools, the visual differentiation and merge facilities should be generic across any diagram type. Finally, we wanted to seamlessly add version control, diagram differentiation and merging to Pounamu using its plug-in API, rather than modifying its code directly.

Much research has been done on the issue of version control, differencing and merging support for programming language editors and other (textual) document editors. Various version control tools have been developed and made available as remote services and plug-ins for many IDEs, such as CVS [7] and RCS [21]. More complex versioning facilities are supported in some specialized program editors, such as Mjolner [14], making use of abstract syntax graphs to link fine-grained versions of artifacts.

Many textual differencing tools share the approach of diff [11] and related tools, determining a set of additions and deletions to change file A into file B [10], [15]. However, extensions to this have been made to support binary and, more recently, XML file differencing, such as the IBM and Stylus Studio XML Diff/Merge tools [12], [19]. Several IDEs with diagramming tools provide model-based differencing, some using custom approaches and others XML-based model differencing [17] [16], [18]. The approach of Ohst et al [17] use a model of drawings and version histories to detect changes and present to users via diagram colour annotation. Several tools [16], [18] use XML differencing of the model structures to detect changes and support graphical annotation of the XML to indicate changes between compared versions. They provide interactive user acceptance or rejection of changes. While comparing design diagrams at the model level has advantages of reuse of diff-style differentiation and merging tools, presentation of the differences textually does not give a very satisfactory sense of actual diagram comparison to the user.

Work has been done providing custom differencing algorithms in software tools, such as architecture design environments. These however tend to use customized algorithms specific to one graphical model type rather than a general approach [25], [22], [17]. Generally there is poor support for differentiating graphical objects such as UML diagrams at the visual, diagrammatic level. Two exceptions are IBM Rational Rose [13] and Magic Draw UML 9.0 [16]. Both tools convert their diagrams into hierarchical text and then perform differentiation on this hierarchy. Changes are shown using highlighting schemes on the text. The main drawback of this approach is that changes are no longer visible in graphical form and thus more difficult to comprehend. The only tools that we are aware of that present changes graphically are a prototype UML editor built at GroupLab [20], and a UML diagram differencing tool we developed in earlier work that leveraged change event objects passed between users' CASE tools to form a delta capturing version differences [5]. This highlighted changes between diagrams by annotation, presenting version differences graphically.

3. OUR APPROACH

Our aim in this work was to extend our Pounamu meta-tool to better support asynchronous collaborative work with diagrams. Key requirements were supporting versioning, differentiation and merging across any diagram type using a set of plug-in extensions. We wanted to present diagram differences graphically and allow users to interactively select differences to accept between versions. Figure 2 shows our approach to developing this generic Pounamu diagram version, differentiation and merging support.

Users create Pounamu diagrams and may check these into a remote CVS server. This is facilitated via a plug-in added to Pounamu to support check-in/check-out to a CVS server Another user may check out a Pounamu diagram from a CVS repository (1). If another user currently has a copy of the diagram, an alternate version is created enabling both to modify it. This new version of the diagram is then modified by the developer using Pounamu's graphical editing tools (2). The developer may then make the modified version accessible to others by checking it back into the CVS repository (3). Another user, for example the original diagram creator, may check this alternate version of the diagram out (4) and then apply the differencing plug-in to compare changes between the two versions of the diagram. Our Pounamu diagram differencing plug-in decomposes the XML describing a model project into a Java object graph. These Java objects are then compared to identify differences. The differences are translated into Pounamu Command objects, each of which embodies a set of API calls that describe editing events that take place in a Pounamu model project (e.g. add shape, connect shapes, move shape, set shape property, delete connector etc) (5). These generated Pounamu Command objects represent the set of changes that need to be made to one diagram version to convert it into the other.

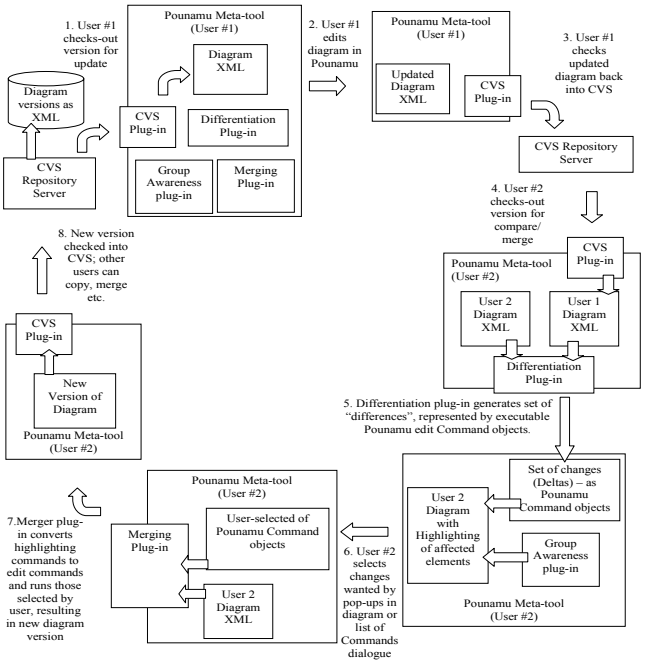

Figure 2. Our Approach to supporting generic diagram versioning, differentiation and merging in Pounamu.

In earlier work we have developed group awareness facilities for Pounamu as part of the research carried out to add plug-in collaborative editing support [9]. These facilities enable us to provide awareness information to users while they are collaboratively editing diagrams by highlighting other users' additions, modifications and deletions to a diagram in near-real time. As part of creating the group awareness component for Pounamu we developed a core set of highlighting schemes and an

API to the plug-in for depicting changes in a visual diagram. One of our goals while developing the visual differencing and merging plug-ins for Pounamu was to reuse the highlighting support provided by our group awareness plug-in to graphically decorate one version to highlight the differences between the diagrams for the user (6). The user is then able to see in the diagram view in Pounamu differences between model project versions and can interactively select the changes to accept or reject (7). Accepting a change causes its associated Pounamu Command to be executed, updating the diagram and thus providing selective merging of diagrams. The merged diagram can be checked back into the CVS repository for others to use (8).

4. ARCHITECTURE

Each Pounamu model project consists of a set of model entities and associations. A number of diagrams, or views, are provided of the model entities and associations allowing users to view and edit the model information. Each view contains a number of shapes and connectors. Shapes are linked by connectors (with owning parent and child shape). Connectors may be visible e.g. lines between shapes, or invisible e.g. representing containment of a set of shapes by another, layout constraints, etc. Every connector or shape has a number of attributes. Each shape has a unique persistent identifier, an objectID, and also a "rootID", which is the objectID of the root version of a shape object. Shapes in different versions can be identified uniquely by their objectID and different versions of a shape derived from the same root shape are identified by sharing the same rootID value.

We represent Pounamu model elements and view elements as XML files for storage, as shown in Figure 3. Our experience developing and evaluating various collaborative model-view based diagramming tools has shown that model differencing alone is insufficient to effectively support diagram editing, differencing and merging [5], [6], [9]. We need to difference diagram-level data and by merging detected changes, the underlying model is also updated.

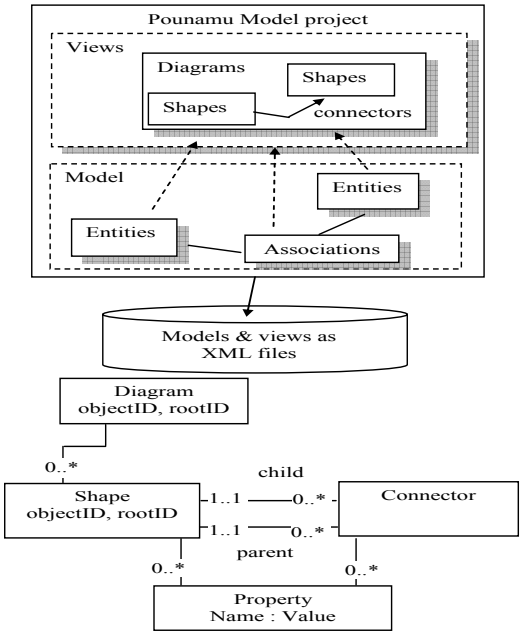

Figure 3. Pounamu model projects, views and the generic Pounamu diagram data structure.

Figure 4 illustrates the key architectural components and their interactions in our extended Pounamu environment. A CVS client plug-in is used to check-out a diagram version (1), which is then converted from its XML save format into Java objects (2) and then rendered and displayed to the user. Edits to the diagram (3) update the internal Java object structure. When comparing diagram versions, an alternative version of the diagram is retrieved (4) and the two versions compared by our diagram differentiation plug-in (5). This generates a set of Pounamu Command objects to describe the changes (delta) between the two diagram versions (6). The diagram highlighting plug-in from our synchronous collaborative editing system for Pounamu is used to highlight changes in the diagram (7). A diagram merging plug-in uses the Command objects to update one diagram version (8), generating a new, merged version for check-in to the CVS repository (9).

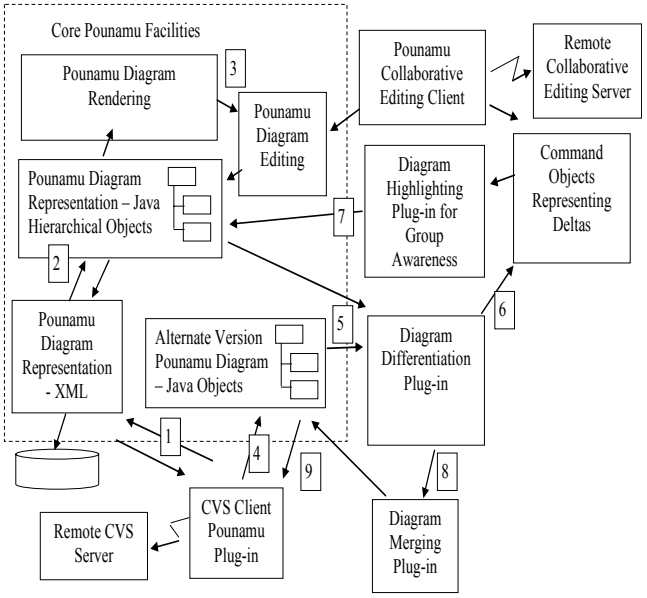

Figure 4. Architecture of the extended Pounamu Environment.

Figure 5 presents pseudo code describing our view differencing algorithm. Differences between views are determined by iterating through all views in a Pounamu model project and comparing each view with corresponding views in another version of the model project. We generate a list of editing Command objects that, when run on one version, convert it to the other version. The set of Command objects generated correspond to line inserts, updates and deletes in a CVS-style textual differencing algorithm.

Let us assume we are comparing two different versions of a certain Pounamu view, view1 and view2 respectively. We compare these two alternative versions of the same view on the basis of shapes and connectors present in each view. We build up a set of Pounamu Command objects that encapsulate the changes that would need to be made to view2 to convert it to view1. The two operations, diffShapes() and diffConnectors(), are used to identify the differences between two views. The operation highlightChanges() is used to iterate over the change list (made up of generated Command objects) and highlight one version to indicate differences to the user.

```
/* Differentiate two different views*/
diffViews(view from project 1, view from project 2) {
     diffShapes(view1, view2, changesList)
     diffConnectors(view1, view2, changesList)
     highlightChanges(view2, changesList)
 }

 /* Differentiate relating to shapes in a particular view.*/
diffShapes(view1, view2, changesList) {
     Vector view1PounamuShapes = view1.getShapesVector();
     for all (existingShape : shapes in View1) {
          /* Get shape in second view with the same shape root id */
          PounamuShape secondViewShape = view2.findShape(existingShape.getRootID());
          if ( secondViewShape != null ) { /* if same-rooted shape exists... */
               /* Check if all attributes match.  If same do nothing; else find differences */
                    if (do the two shapes have the same position) {
                       /* If false add to shapes that have been moved */
                       changesList.add(MoveShapeCommand(secondViewShape, dx, dy));
                    } /* Check if shape has been Resized */
                    if (are the two shape of the same size) {
                      /* If false add to shapes that have been resized */
                      changesList.add(ResizeShapeCommand(secondViewShape, dwidth, dheight));
                    } /*Attributes Changed */
                    if (check if shape attribute values are the same) {
                      for all ( prop : props of existingShape where oldValue=existingShape.getValue(prop) !=
                                newValue=secondViewShape.getValue(prop)
                           changesList.add(ChangePropertyCommand(secondViewShape, prop, oldValue, newValue));
                    }
          } else {
               /* shape does not exist in view2 so it has been deleted… */
               changesList.add(DeleteShapeCommand(secondViewShape));
          }
     }
     Vector view2PounamuShapes = view2.getShapesVector();
     for all ( secondViewShape : shapes in View2 ) {
          /* Get shape in other view with the same shape root id */
          PounamuShape existingShape = view1.findShape(secondViewShape.getRootID());
          if ( existingShape == null ) {
               /* shape does not exist in view1 so have added it */
               changesList.add(NewShapeCommand(view1, secondViewShape.getType(), secondViewShape.getRootID()));
               for all ( prop : properties of secondViewShape )
                    changesList.add(ChangePropertyCommand(newShape,prop,null,secondViewShape.getValue(prop)));
          }
     }
 }

 /* highlight changes in view */
 highlightChanges(view2, changesList) {
     for all ( change : changes in changesList) {
          if ( change == NewShapeCommand)
               shape = NewShapeEvent.getShape();
               shape.setTempProperty("oldLineColor", shape.getLineColor()); // remember old values...
               ...
               shape.setLineColor(red);
               shape.setBorderThickness(2);
               …
          else if ( change == MoveShapeCommand )
          ...
     }
 }
```

Figure 5. Pseudo Code Describing the Model Project Differ.

As views are made up of shapes and connectors they are compared on this basis. All Pounamu views, shapes and connectors have a unique, persistent object ID (a GUID) generated when they are created, making them uniquely identifiable across users' model projects. View elements are similarly tagged with a unique object "root ID" in addition to their own unique object ID. If a view is created from an existing view i.e. an alternative version or revision, the new view's elements are tagged with the root ID of the elements they are derived from. If a view is created from scratch, each element's root ID is the same as its object ID.

When comparing views the root ID is used to identify view elements with the same common root object i.e. elements in each view that are alternative versions of each other. This can be thought of as a simplified form of origin analysis [24] and is a way of implementing object uniqueness [17]. Connector alternative versions are identified by their source/target shape root IDs. We use a two-pass approach over the diagram's shape and connector elements rather than graph-based traversal [25] to generate the Command objects representing a delta between them. Firstly we iterate over each shape in view1 and see if it exists in view2. If not, the shape must be deleted to convert view1 to view2 (which will result in all of its connectors also being deleted). A delete shape Command object is generated to represent this action for each shape in view1 but not present in view2. Shapes present in both view1 and view2 have their size, location and other properties compared. Editing commands are generated to modify any non-matching properties. We then look for shapes in view2 not present in view1. Any found must be added to view1 to convert it to view2, so shape addition Commands are constructed and shape initial size, location and property value setting commands are generated to initialize the newly added shape. We then pass over the connectors using a similar approach: locate connectors present in both views and generate property change Commands to synchronise their properties; generate connector delete Commands to remove connectors in view1 not in view 2; and generate connector Create and property initialization Commands to add connectors that are in view2 but not in view1. We have found this two-pass approach to be sufficient for Pounamu views and generally less complex to implement and more efficient on large views than graph traversal approaches.

The diffShapes() method iterates through all shapes in a view and tries to find shapes with the same root ID in the view that it is being compared to. If the shape can't be found, it either needs to be deleted or added to synchronize the two views. If the shape

exists then it checks for differences in a shape's size, location and attribute values. All differences are recorded as Pounamu Command objects (NewShapeCommand, ResizeShape Command ChangePropertyCommand etc). These Pounamu Command objects may be executed in order to synchronise the two views which results in view1's diagram data structure being converted into the same as view2's. Any newly added shapes in view1 have their rootID set to the rootID of their view2 shape they have been derived from, allowing identification of common ancestors for shapes. Complex shape structures e.g. containment and layout-based constraints are supported by our comparison and merging mechanism as they are driven by change events generated by the Command object execution.

The Diagram Highlighting plug-in from our collaborative diagram editing client provides a highlightChanges() function. This decorates the graphical diagram rendering to indicate changes required to convert one version to the other. The diagram highlighter iterates through the generated Pounamu Command objects and for each modifies the Pounamu diagram elements – bold red for added shapes; dashed fine line around deleted; red fill box for changed shape and connector properties; and dashed origin and line for move/resize of shapes. The highlighter modifies standard properties of shapes and connectors to achieve some of these highlights and adds its own annotation shapes and connectors to the diagram to achieve others. Upon carrying out merging of changes (i.e. execution of editing Commands), standard Pounamu diagram editing event propagation notifies the highlighting component which then removes annotations.

Users may select a subset of Commands to apply to a view to effect merging of changes by interacting with the decorated diagram elements. Accepting a change and having its associated Command object executed results in a change-by-change partial merging. This may result in some commands not being able to be applied. For example, a NewShapeCommand is not applied meaning a subsequent ChangePropertyCommand on the shape not being able to be applied to the view. Similarly during merging semantic errors can be produced e.g. two classes with the same name or an invalid type association between objects. We allow the Pounamu tool's semantic constraint handling to detect and highlight these as merging proceeds. However, applying the Command objects (i.e. modifying the syntactic structure of the view) and providing the user a list of semantic errors introduced into the new version could be supported in advance of user-acceptance of syntactic merge changes.

5. EXAMPLE USAGE

We illustrate our diagram differentiation and merging algorithm with a simple class diagramming tool example. Two colleagues, John and Tim, are working with a Pounamu-created UML design tool at different locations. By using Pounamu versioning capabilities they can asynchronously collaborate on a Pounamu model project. In order to find differences between different versions of model project's views they are provided with our CVS version control plug-in, diagram differencing plug-in and difference merging plug-in.

A CVS repository enables storing and versioning of binary and text files and sharing of these files among distributed users. Asynchronous collaboration in Pounamu is facilitated by versioning Pounamu model projects and their views using a shared CVS repository via a plug-in to the Pounamu modeling tool. This CVS plug-In enables storing and retrieving Pounamu model project information and individual diagram information in their XML save file format from a shared, remote CVS repository. Users who wish to collaborate on a model project asynchronously are expected to check in (store) their model projects into a CVS repository. Remote collaborators can check out (retrieve) model projects thus enabling asynchronous collaboration.

Let us assume John creates a new UML class diagram for an existing Pounamu UML tool model project that he and John are working on. John adds shapes and connectors, moves and resizes shapes, sets various properties and so on to create his class design diagram. In order to share these changes with Tim, John "checks in" his model project with the CVS repository. The current state of the new class diagram checked in by John is illustrated in Figure 6.

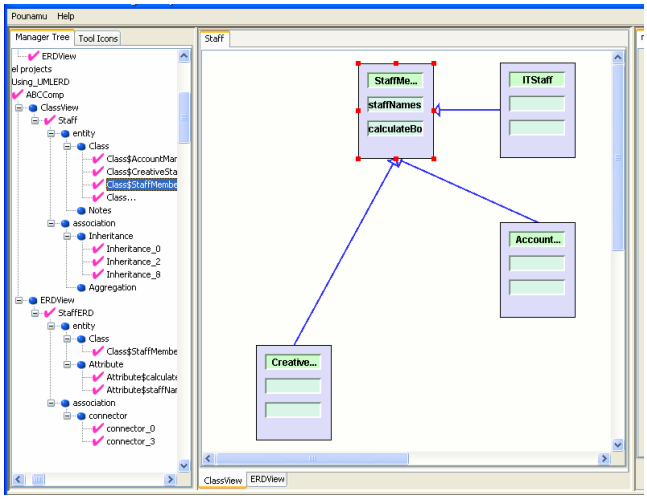

Figure 6. Initial class diagram view as checked in by John.

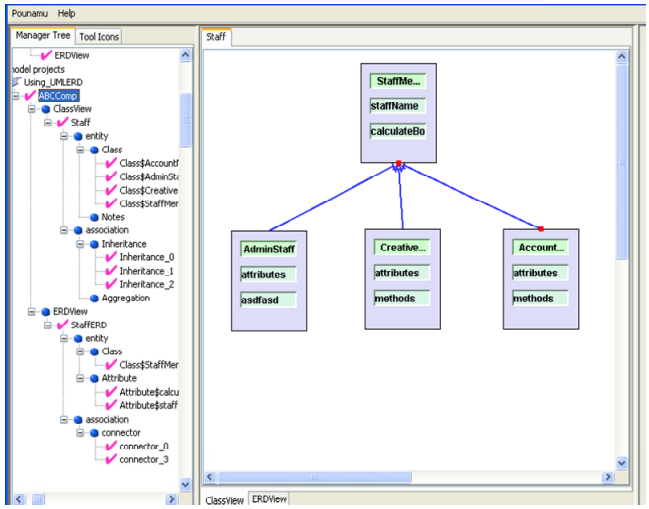

Figure 7. Tim's alternative version of the class diagram.

Tim subsequently checks out the model project containing this same UML class diagram from the CVS repository, thereby creating an alternative version of it and asynchronously makes changes to his version of the class diagram. Tim's alternative version of the diagram after making several changes is shown in Figure 7. Class icon shapes have been moved, deleted and some

of their properties changed; association connectors have been added and deleted. Tim the checks his model project back into the CVS repository, resulting in his alternative versions of changed diagrams being checked in as well.

In order to see any changes that might have been made by Tim, John differentiates his model project against what Tim has checked in. Appropriate pop-up menu items in the right-hand Pounamu tree viewer are provided when the CVS versioning and diagram differentiation plug-ins have been enabled. On selecting the "Diff with Later Versions" menu item for an open diagram view, John is presented with a differentiation dialogue box as show in Figure 8. John chooses "Version 1.5" checked in by Tim to differentiate his current version ("Version 1.4") of the class diagram against. On executing the differentiate command Tim's version is checked out of the CVS repository in read-only mode, and the differentiation algorithm from Figure 5 is applied comparing Tim's version (view 1) with John's version (view 2) of the diagram. A set of Pounamu Command objects are generated representing the delta between Tim's alternative version and John's version of the class diagram. John's diagram is then annotated to show the differences by our diagram highlighting plug-in iterating over the generated Command objects.

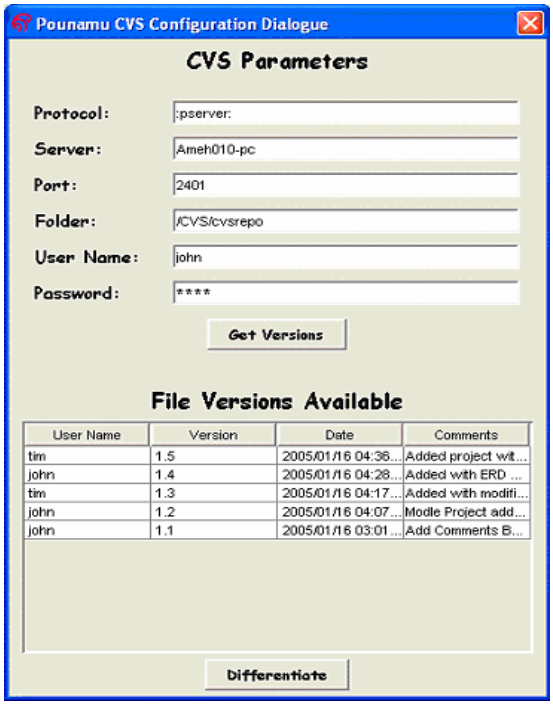

Figure 8. Differentiate Dialogue Box.

Figure 9 shows the highlighted differences between Tim and John's versions of the class diagram. Solid lines denote creation of a shape or an association (e.g. "AdminStaff" and its association to "StaffMember"). Dotted lines denote deletion of a shape or association (e.g. "ITStaff" and its association to "StaffMember"). Shape movement is donated by an empty dotted box pointing to the new place where the shape has moved (e.g. "CreativeMember" repositioning). The background colour of the view has changed to a darker shading to enable users to be able to view the extent of changes to a view at a glance. A white background denotes no change while a very rich pink background denotes a large number of changes. Dark highlighting of entities and attributes denote creation and modification of values. A list of all changes is also available via a dialogue.

A user is free to accept or reject any changes presented. Pop-up menu items are provided with each change to enable John to accept or reject each of the changes made by Tim as he requires. After careful analysis John has decided to accept the shape and association created while rejecting the shape and association deletion. The resulting view that John sees after accepting and rejecting all change is shown in Figure 10. However note that some changes are dependent on previous changes made e.g. accepting the setting of AdminStaff properties and creation of its association to the StaffMember class requires firstly accepting the creation of the new AdminStaff class icon. If this creation has not been accepted, the subsequent changes are marked as "not able to apply".

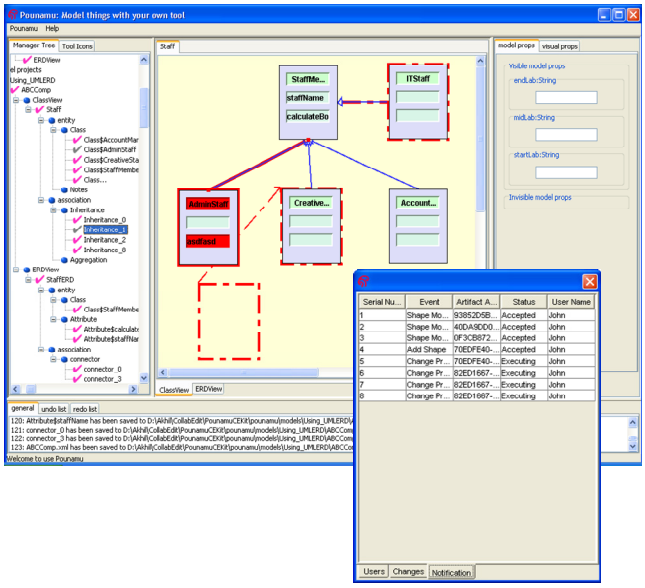

Figure 9. View after Differentiation and highlighting.

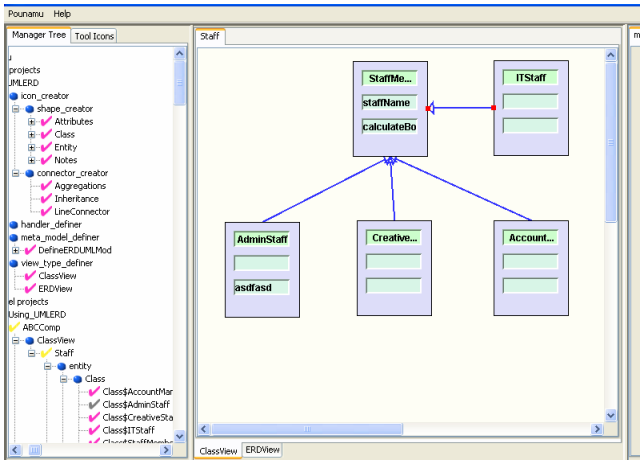

Figure 10. Merged document to be checked in by John.

6. DESIGN AND IMPLEMENTATION

One of the major design goals was to make no changes to the existing single-user Pounamu code when adding our CVS, diagram differentiation and diagram merging capabilities to

Pounamu. Our plug-ins were designed to use a Service Oriented Architecture [9] where each service, collaborative editing, group awareness, and version control, are discovered at run-time by Pounamu environment client plug-ins. The diagram differentiation plug-in was the major development. Design of the visual differentiation tool was made easier by translating differences detected between diagram versions into Pounamu editing Command objects. We then reused the diagram highlighting plug-in from our group awareness support plug-in to highlight a view using these command objects. Version merging was supported by allowing the user to selectively run all or some of these Command objects on the target view, producing a merged view version.

Figure 11 shows the design and interactions of our versioning, differentiating and merging plug-ins. Assume that user John decides to differentiate his existing model project with a later version of Tim's stored in the CVS repository (1). On retrieving Tim's checked in model project as an XML document (2) and having its views loaded into Pounamu's object structures (3), John may differentiate the two (4) using the Differentiation plug-in. A set of Pounamu Command objects (5) are generated by the Differentiation plug-in's diffViews() method, by traversing the view structure of view 1 and comparing each shape and connector to their equivalents (if they exist) in view 2.

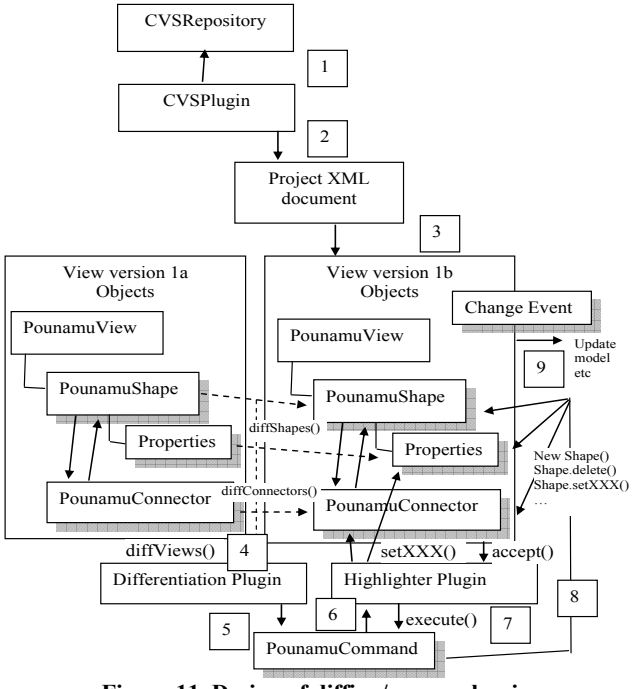

Figure 11. Design of diffing/merge plug-ins.

The sequence of events that take place within diffShapes() and diffConnectors() are quite similar. When diffShapes() is called it retrieves a list of PounamuShape objects in the form of a vector from the first PounamuView (view 1). It then uses this list for comparison. Iterating through the list of PounamuShapes it obtains the rootID for each shape. A rootID uniquely identifies each shape. The rootID is used to retrieve similar objects for the previous version. If the object is not found a NewShapeCommand and a number of SetPropertyCommands are added to the changes list, denoting the need to add this shape if the two versions are to be synchronised. Similar comparisons are carried out for removal, movement, resizing and changes in shape or connector object properties. The changes list containing the set of PounamuCommands needed to fully convert view 1 to view 2 is then passed to the Highlighting plug-in from our group awareness system (6).

The Highlighting plug-in examines each Command object and sets various visual properties (colour, line thickness and style, border, shadow position and arrow etc) of view 2's Pounamu view objects. The view is re-rendered once this is complete, decorating the view to indicate the version differences. A menu item for each Command object is added to each Pounamu view object's pop-up menu, allowing the user to selectively accept or reject the difference. Accepting the difference tells the Highlighter to run the Command, by calling its execute() method. For each kind of Command, various changes are made to the view e.g. add new shape, delete shape, set shape property etc (8). The Pounamu model is updated when the view objects are thus changed (9), updating any other views sharing the model information.

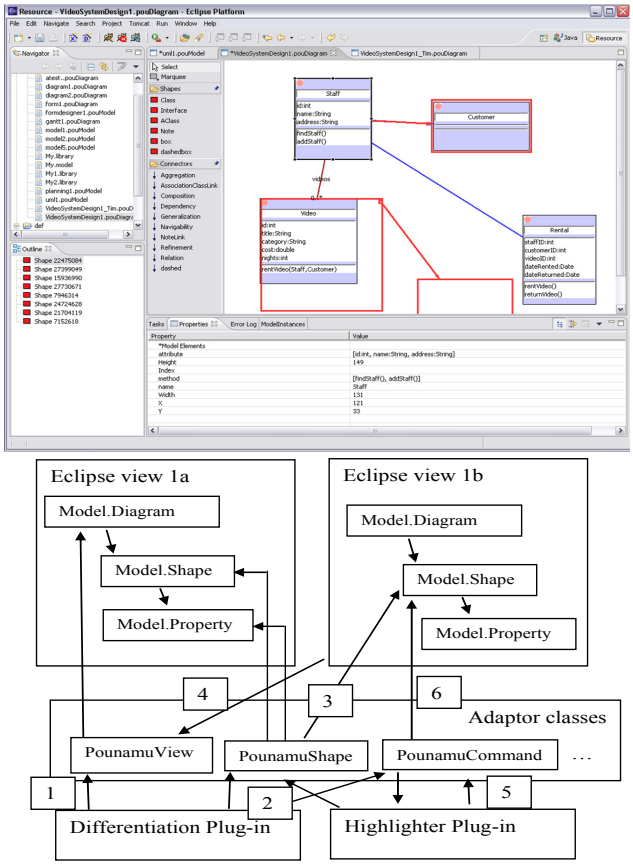

Figure 12. (a) Eclipse Pounamu plug-in differentiation and (b) adaptor objects for the differentiation plug-ins.

We have recently developed an Eclipse plug-in for Pounamu that loads Pounamu meta-tool specifications and provides an Eclipse Graphical Editor Framework (GEF)-based modeling tool. An example of view differentiation with our Differentiation and Highlighter plug-ins is shown in Figure 12 (a). As the Eclipse plug-in does not represent view objects nor Command objects the same way as our Pounamu modeling tool we developed a set of

Adaptor classes to map the methods of our Eclipse plug-in onto compatible named and typed classes of our Pounamu modeling tool. This allowed us to add our Differentiation and Highlighting plug-ins into the Eclipse-based Pounamu environment and support these activities. Figure 12 (b) illustrates the Adaptor objects needed to achieve this. Our Eclipse plug-in does not store the views as XML, hence the need for adaptation to the plug in's internal object structure.

7. EVALUATION

We used a combination of the Cognitive Dimensions framework [4], Gutwin's groupware assessment framework [8], and a user survey to assess the effectiveness of our Pounamu plug-ins for asynchronous diagram version control, differentiation and merging. The Cognitive Dimensions framework provides a generic approach to measuring various usability characteristics of notations and their environments [4]. We were particularly interested in assessing our plug-ins' support for the dimensions of *visibility of changes*, *hidden dependencies*, *viscosity* (i.e. ease of change of content), *error-proneness*, *hard mental operations*, *consistency* and *closeness of mapping* (i.e. how close a representation matches a user's mental model). From our analysis, key results were that a major benefit of our approach is that differences are highly visible, being presented as graphical annotations to one of the compared diagram versions. Unlike approaches using comparison of models or textual versions of diagrams there are no hidden dependencies between the differences presented and accepted by the user and the resulting merge operation. Interactive acceptance of changes in the diagram by the user reduces both error-proneness and hard mental operations during merging of versions. Our approach to presenting changes is consistent, using colour (red) to denote change, though the difference between some change types is minimal e.g. set property vs create vs move shape. The graphical mark-up of changes in a diagram our approach adopts is similar to the way textual and XML differencing tools [2], [19] present changes providing an element of consistency.

Gutwin's framework [8] provides a groupware-specific set of usability and assessment criteria. As our work is an asynchronous groupware extension to Pounamu, we evaluated our differentiation and merging plug-ins using the *presence*, *authorship*, *identity*, *gaze*, *action*, *intention* and *location* criteria of the framework. Key results from this analysis are that users can identify other's presence and authorship of changes as these are annotated with the author's name and represented in the dialogue view of changes (with changes by multiple authors represented differently in the graphical presentation through tool tips). Changes to diagram content are explicitly and clearly represented and user interaction is explicitly with a graphical representation of changes via pop-ups to accept/reject them. Partial support for intention awareness is supported as the differentiated and visualized changes represent another's (intended) actions for the merged diagram.

We conducted a formal user survey of our group awareness plug-ins for collaborative editing [9]. This involved 10 users carrying out a combination of synchronous and asynchronous design tasks with a Pounamu UML tool over multiple sessions. These were mainly UML diagram review, creation, editing and discussion. Users performed various asynchronous UML diagram editing tasks with our versioning, differentiation and merging plug-ins. They also performed synchronous UML diagram editing tasks with other Pounamu groupware plug-ins. Feedback on our asynchronous editing support features was very positive, including their response time, approach to presenting changes, support for incremental change accept/reject and overall support for asynchronous diagram-based design activities. Some users requested control over the way changes are presented by the highlighting plug-in. Some requested the ability to have multi-version merges, like in MS Word, where tracking of changes by several users on the same document is supported with different coloured highlighting.

Most existing diagramming tools with versioning support provide model-based differencing using diff or XML diff-based tools [18], [19]. Those that provide diagram differencing utilize textual comparison of diagram content. For example Rational Rose and Magic Draw convert diagrams into a hierarchical textual representation which is diffed and then changes between diagram versions presented as highlighting schemes on the text. The main drawback of this approach is that changes are no longer visible in graphical form and thus more difficult to comprehend (or introduce *hard mental operations* in Cognitive Dimensions terminology) [20]. In contrast our plug-ins provide in-situ presentation of differences within diagrams and interactive accept/reject by users with immediate visualization of the accepted merged change. The generic nature of our differencing algorithm and use of Pounamu Command objects to represent differences between versions means it can be applied to *any* Pounamu-specified diagramming tool; for example, Figure 1 (c) and (d) show application to a Gantt chart tool generated using Pounamu. As the architecture of our plug-ins uses Pounamu's view representation objects and Command objects to update view content on merging, it is compatible with other Pounamu core features and plug-ins. These include semantic constraint checking via event-driven rules, model-view consistency across multiple diagrams when changes are merged into a diagram, and seamless integration with our synchronous editing plug-ins for Pounamu. In addition, because of this architectural approach we have managed to integrate our differentiation plug-in into our Eclipse modeling plug-in for Pounamu, using the Eclipse plug-in's object adaptors without modification of either.

Our approach has some limitations. As indicated above from our user survey and assessment against Gutwin's framework, users cannot control how changes detected by the differentiation plug-in are presented. As Pounamu is a meta-tool allowing very flexible definition of diagrammatic forms and meta-models, this is somewhat frustrating to users. Similarly, we currently only support batch-oriented comparison of one diagram version to one other, without tracking changes in a diagram across multiple versions or supporting several-version diagram merge. Semantic errors can be introduced easily when merging versions e.g. same-named method or class; type-mismatch, invalid association. Currently we allow Pounamu's constraint mechanism, which is driven by event handlers detecting diagram and model object changes, to detect these and present them using the user-specified mechanism in the meta-tool. Ideally semantic conflicts that may be introduced by accepting a change should be indicated in the annotated diagram similar to syntactic changes. Conflicting syntactic changes e.g. one user has deleted shape while another moves it or sets its properties, are detected by our differentiation algorithm. However no attempt is currently made to re-order

changes or indicate to the user that accepting one change may invalidate another. The scalability of our diagram highlighting approach is limited, with a very complex diagram with a large number of changes resulting in very complex, over-lapping highlighting. We need support for users to see a subset of changes and be able to interact precisely with change highlights in views.

Future work includes providing users with the ability to change the highlighting used by the highlighting plug-in. This can be done using Pounamu's meta-tool to specify individual event handler plug-ins (dynamically loaded Java script) for each highlighting scheme but requires re-engineering our highlighting plug-in. Semantic conflict detection and presentation before and during version merging could be supported using Pounamu event handlers that generate events indicating a conflict. Currently these implement a user-defined conflict presentation strategy themselves via arbitrary plug-in Java scripts. All semantic constraints for a Pounamu tool would instead need to be encoded in a uniform way.

8. SUMMARY

We have developed a set of plug-in components for the Pounamu meta-tool that seamlessly support asynchronous diagram versioning, differentiation and merging. Any diagram type defined with Pounamu may make use of these capabilities to compare versions of the same diagram, the differentiation plug-in generating a set of generic Pounamu Command objects to represent the delta between the versions. A reused view highlighting plug-in visually annotates the diagram to indicate the changes between versions and supports interactive, selective accept/reject of changes by the user. By use of a set of adaptor classes our plug-ins provide similar support within Eclipse graphical editors derived from Pounamu meta-tool specifications.

9. REFERENCES

[1] Conradi, R. and Westfechtel, B. Version Models for SoftwareConfiguration Management. ACM Computing Surveys, vol. 30, no. 2, p. 232-282,1998.

[2] Eclipse Version Tree plug-in for CVS, http://versiontree.sourceforge.net/

[3] Ellis, C.A., Gibbs, S.J., and Rein, G.L., Groupware: Some Issues and Experiences, CACM, vol. 34, no. 1, 1991.

[4] Green, T. R. G., Burnett, M. M., A Ko, J., Rothermel, K. J., Cook, C. R., and Schonfeld, J., Using the cognitive walkthrough to improve the design of a visual programming experiment, 2000 IEEE Conf. on Visual Languages, pp. 172-179.

[5] Grundy, J.C., Hosking, J.G. Mugridge, W.B. and Amor, R. Support for collaborative, integrated software development, 7th IEEE Conf. on Software Engineering Environments, Nordwijkerhout, The Netherlands, 4-5 April 1995.

[6] Grundy, J.C., Hosking, J.G. and Mugridge, W.B. Inconsistency management for multiple-view software development environments, IEEE Transactions on Software Engineering, vol. 24, no. 11, November 1998, 960-681.

[7] GNU, CVS - Concurrent Versions System, ww.gnu.org/software/cvs

[8] Gutwin, C. and Greenberg, S., A Descriptive Framework of Workspace Awareness for Real-Time Groupware, Computer Supported Cooperative Work, vol. 11 no. 3, 2002, 411-446.

[9] Mehra, A. Grundy, J.C. and Hosking, J.G. Adding Group Awareness to Design Tools using a Plug-in, Web Service-based Approach, 6th International Workshop on Collaborative Editing Systems, CSCW, Nov 2004, Chicago.

[10] Heckel, P. A technique for Isolating Differences Between Files, CACM, vol. 21, no. 4, April 1978, pp. 264-268.

[11] Hunt, J.W., and McIlroy, M.D., An Algorithm for Differential File Comparison., Computing Science Technical Report No. 41, Bell Laboratories, 1975.

[12] IBM. XML Diff and Merge Tool. http://www.alphaworks.ibm.com/tech/xmldiffmerge.

[13] IBM, IBM Rational Software, http://www-306.ibm.com/software/rational/.

[14] Magnusson, B., Asklund, U., and Minör, S., "Fine-grained Revision Control for Collaborative Software Development ," 1993 ACM SIGSOFT Conf. on the Foundations of Software Engineering, Los Angeles CA, Dec.1993, pp. 7-10.

[15] Miller, W. and Myers, E.W., A File Comparison Program. Software - Practice and Experience, vol. 15, no.11, November 1985, 1025-1040.

[16] No Magic Inc., 9.0 ed: MagicDraw UML, 2005.

[17] Ohst, D., Welle, M. and Kelter, U. Difference tools for analysis and design documents, 2003 IEEE Conf. on Software Maintenance.

[18] Sparc Systems, The Compare Utility (Diff), http://www.sparxsystems.com/resources/diff/.

[19] Stylus Studio, XML Diff tool, http://www.stylusstudio.com/xml_diff.html.

[20] Tam, T., Greenberg, S. and Maurer, F., Change Management, Western Computer Graphics Symposium, Panorama Mountain Village, BC, Canada, 2000.

[21] Tichy, W. F. 1985. RCS-a system for version control. Software-Practice & Experience, vol. 15, no. 7, 637-654.

[22] van der Westhuizen, C. and van der Hoek, A. Understanding and Propagating Architecutural Changes, 3rd IEEE/IFIP Conference on Software Architecture, pp. 95-109.

[23] Zhu, N., Grundy, J.C. and Hosking, J.G.. Pounamu: a meta-tool for multi-view visual language environment construction, 2004 IEEE Conf. on Visual Languages and Human-Centric Computing, Rome, 25-29 Sept. 2004, 2004.

[24] Zou, L. and Godfrey, M. Detecting Merging and Splitting using Origin Analysis, 10th International Working Conference on Reverse Engineering, Victoria, B.C., Canada , 11-13 Nov, 2003.

[25] Zündorf, A., Wadsack, J.P., and Rockel, I. Merging Graph-Like Object Structures. 10th International Workshop on Software Configuration Management. 2001.