Engineering for Human-Computer Interaction, Chatty, S. and Dewan, P. Eds, February 1999, © Kluwer Academic Publishers.

Engineering Component-based, User-configurable Collaborative Editing Systems

John Grundy
Department of Computer Science, University of Waikato, Private Bag, Hamilton, New Zealand
jgrundy@cs.waikato.ac.nz

Abstract: The ability to collaboratively edit work artefacts is important in many kinds of editing tools, including Computer-Aided Design (CAD) tools, Computer-Aided Software Engineering (CASE) tools, drawing packages, and document editors. However, most existing such tools either do not support collaborative editing or provide limited collaborative editing facilities. We describe our recent work in adding collaborative editing support onto a previously single-user CASE tool, using a component-based approach. Our collaborative editing components allow users to move from asynchronous to synchronous editing as desired, and even allow a user to support different levels of collaborative editing with different other users simultaneously. Major advantages of our approach include no changes to the implementation of the component-based CASE tool, nor the collaboration-supporting components, were necessary. Additionally, our components that facilitate collaborative editing are readily reusable in other tools adopting a similar component-based software architecture.

Keywords: groupware, component-based software, CASE tools, reusability

1. INTRODUCTION

As cooperating workers have become increasingly distributed in geography and time, support for collaborative editing has become increasingly important in many editing tools. For example, users of software development tools generally require facilities to support asynchronous work, e.g. version and configuration management tools and merging capabilities. They also often desire synchronous editing

capabilities, which allow for example designers to closely collaborate on evolving systems, debugging to be done co-operatively, or documentation and reverse-engineered designs to be discussed. Users may also want editing capabilities in between these extremes. For example, being kept aware when other users modify a shared design but not having their own design modified.

Unfortunately most existing editing tools do not support this range of collaborative editing facilities for the same kind of work artefact, or only support one particular approach for different kinds of artefacts. It is also often difficult to seamlessly move between different "levels" of collaborative editing as desired. An additional problem from a tool engineering perspective is that almost all editing tools have to be specifically engineered to support even one type of collaborative editing. It is usually a large effort to retrofit such capabilities onto existing tools, and such modifications are often infeasible from an organisational or engineering perspective.

We describe our recent work in adding user-configurable collaborative editing facilities onto an object-oriented CASE tool. We have done this using a software component-based approach i.e. we have developed software components which support collaborative editing facilities and "plugged" these components into our CASE tool without modifying the CASE tool implementation or architecture in any way. These collaborative editing components can also be plugged into other editing tools using a component-based software architecture, without necessitating any change to the components nor the tool.

2. COLLABORATIVE EDITING REQUIREMENTS

The screen dump in Figure 1 is from the JComposer object-oriented CASE and meta-CASE tool (Grundy et al., 1997; Grundy et al., 1998). JComposer provides editable graphical and textual views of object-oriented and component-based software systems, supporting their specification, design, implementation, documentation and reverse-engineering. It also supports generation of CASE tools based on a component-based software architecture and framework. The diagram ("view") in Figure 1 shows a process support tool repository being specified. JComposer will subsequently generate a component-based implementation of this graphical specification. Other views can be developed which specify e.g. multiple views of a process model as graphical and/or textual editors (Grundy et al., 1997).

Users of JComposer often wish to collaborate to develop specifications and designs, as well as to implement, debug and reverse-engineer systems. For example, collaborating developers may want to synchronously edit a view to closely collaborate in building a specification or design. When one user makes changes to the view at this close "level" of collaboration, the exact same change is made to the other user's view. Sometimes users will want to have a description of changes made shown to collaborators, rather than automatically actioned. They can then decide whether to have the change made immediately to their version of the view, whether to discuss the change further, or whether to reject it. At other times developers will work asynchronously, not having changes broadcast to others' working on different versions of the same view. Subsequently developers will exchange modifications and merge some or all of them with their own changes to a view. Users may simply request they be informed when some change or sequence of changes are made to a

view they are interested in, with some automatic processing to take place (e.g. to record the change(s), to inform them of the change(s), etc.).

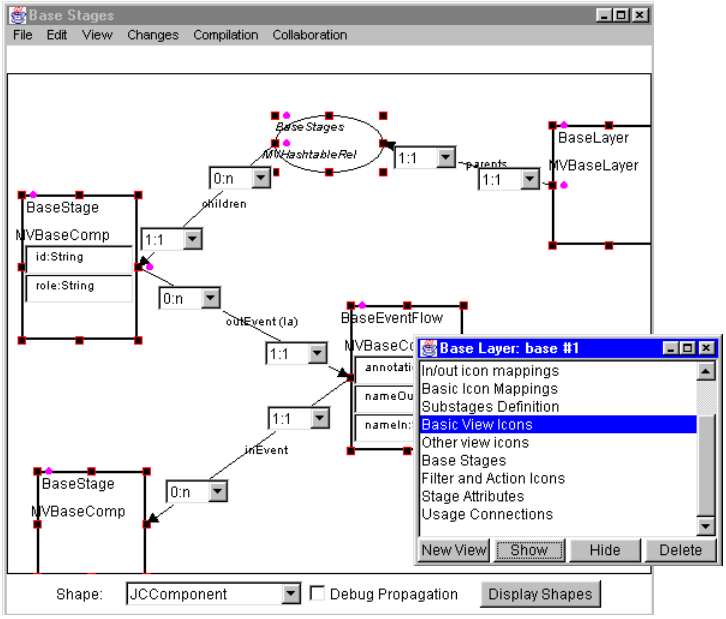


Figure 1. Example of an editable view from the JComposer environment

It has been our experience with developing tools such as JComposer that all of these levels of collaborative editing are required for all kinds of views of software development, at one time or other during the software development lifecycle. Thus tools like JComposer should allow any view to be edited in any of the above levels of collaborative editing, but moreover users should be able to easily move between any of these levels as desired.

JComposer was originally developed with no specific multi-user editing capabilities, but does use a component-based software architecture, allowing new software components to be "plugged into" the environment without modifying existing components. We could have substantially modified the tool to support the kinds of collaborative editing facilities outlined above, but instead chose to take a longer-term view, as we wanted to be able to leverage these kinds of facilities in other editing tools. Thus we required an engineering approach which would, ideally, involve no modifications to JComposer or its software architecture, but instead utilise a component-based approach with reusable collaborative editing components plugged into a tool if required. Advantages of this approach include: ability to reuse collaborative editing components without modification; ability to add collaborative editing to tools without modification; and upgradeable components, i.e. unplugging components to add improved ones.

3. A COMPONENT-BASED SOFTWARE ARCHITECTURE

3.1 3.1. JViews Software Architecture

JComposer is built using a component-based software architecture called JViews (Grundy et al., 1997). Component-based systems are comprised of units of data and functionality, which are composed to build a complete software product. The difference between component-based systems and more conventional software systems (e.g. using function libraries and class frameworks) is that component-based systems allow components to be "plugged" in at run-time, or unplugged and interchanged with other components with comparable interfaces and functionality. This supports user-configuration of systems, reusbility of components, and a more versatile and potentially robust "building block" approach to system architectures.

JViews is built on top of the Java Beans componentware API of Java 1.1 (Javasoft, 1997). The basic structure of a JViews component-based system is illustrated in Figure 2. Components (rectangles) are linked by relationship components (ovals) or simple reference links (solid lines). When a component undergoes a state change, it sends a "change description" object describing this change to "interested" linked components and relationship components. Interested components choose to listen before and/or after the state change occurs, or can even listen when other components receive change description objects. Change descriptions can be stored and used to implement a wide range of system functionality, including undo/redo for diagrams, attribute recalculation and constraint enforcement, versioning and collaborative editing (Grundy et al., 1996).

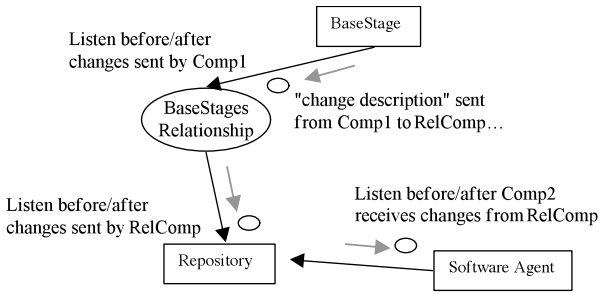


Figure 2. The JViews component-based software architecture

JViews-based environments support multiple views of work artefacts via "view relationships" between repository components and view components. View components are rendered in graphical or textual forms, and provide appropriate editing functionality. Ideally all JViews-based environments should provide the collaborative editing capabilities described in the previous section. However, we did not build such capabilities directly into JViews. In order to support such capabilities, each view in a JViews-based system, such as JComposer, needs additional components "plugged" into the view i.e. components supporting distributed, multi-user editing are connected to JComposer view components. These collaborative editing components "listen" to change descriptions (i.e. editing events) generated by

components in the view, and these changes propagated to collaborating user's environments. When an environment receives such changes, the changes must be forwarded to the appropriate view and actioned as appropriate to support the level of collaborative editing required.

3.2 Collaborative Editing Components

Figure 3 shows some existing JComposer view and repository components and links, for the process model repository in Figure 1, coloured grey. The JViews-based collaborative editing components we have developed are shown in black. The main collaborative editing component is the "collaboration menu", which provides the user interface to configuring the level of collaborative editing on a view, and handles change propagation to/from other users' environments. Each JComposer view has an instance of this component connected to it, and this collaboration menu component "listens" before and after any changes are made to the view. Listening for all view component changes both before and after they are made allows the collaboration menu component to implement locking protocols for synchronous editing, and to lock out changes produced by other users when at a synchronous level of collaborative editing on this or other views. This is necessary as our Java implementation of these collaborative editing components uses multithreading, with separate execution threads handling receiving and actioning of other users' edits.

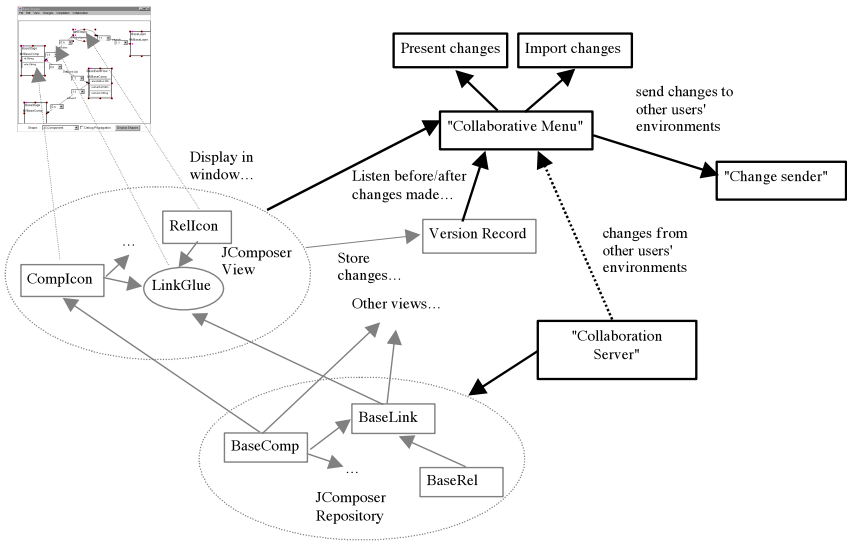


Figure 3. Example of adding groupware components to JComposer

Each JViews view component has a "version record", storing changes made to the view to provide a modification history and to support undo/redo of all view changes. The collaboration menu component listens to this version record, and before a change description is stored in the version record, the collaboration menu annotates it with the name of the user from whom the change originated. After the

change has been stored in the view's version record, it is propagated to other collaborating users who are collaboratively editing the view. We propagate the changes stored by the view in its version record, not all change descriptions generated by the view's components. This is because some view component state changes are caused by the actioning user editing, which we want to propagate and record, but others are caused by these edits (e.g. resizing a line if one of its connected icons is moved). Thus only the *initiating* changes need be stored and propagated, as these "follow-on" changes will be made by the other user environments' editor semantics.

3.3 Change Sending

Changes are sent to collaborating users' environments via a "change sender" component, which runs in a separate execution thread to the collaboration menu and JViews view components. This multi-threading ensures that no editing performance loss occurs for the user of the environment while changes are propagated to collaborators i.e. no blocking of user I/O occurs. The change sender queues change descriptions to be broadcast to other users' environments, and uses point-to-point communication with the other user environment's "collaboration server" components to send the change descriptions. We used the JViews change description serialisation mechanism to serialise change description objects into byte streams and to send them via socket connections to other users.

A "collaboration server" component is attached to each JViews environment's repository component, and allows multiple client socket connections from other collaborators' "change senders". The collaboration server, on receiving change descriptions, forwards them to the appropriate view's collaboration menu component, which deserialises the broadcast change description and actions it appropriately. The collaboration menu has a version record to present changes in (if at "presentation" level), and a version record to import changes into (when at an asynchronous editing level). Figure 4 illustrates the propagation of change descriptions between users' environments. We used this multi-point broadcasting model, rather than relying on a single-server model to provide efficient, robust collaborative editing capabilities. However, a major advantage of our component-based approach is that we could replace the change sender and collaboration server components with ones that implement e.g. an Remote Method Invocation (RMI)-based approach to propagating changes, rather than use sockets, or use a shared single-server architecture.

An additional lightweight "registration" server is provided for registering user names and the user's host and port numbers (to establish socket connections between change senders and collaboration servers). This also allocates unique component ID numbers for each user as they are required. The collaboration menu uses sets of unique ID numbers generated by this server to uniquely tag every JViews view component with an ID number. Each user then has a their own version of view data, with copied view components uniquely numbered, rather than have users distributively share the same component data. This replicated data approach once again provides a very robust implementation, but more importantly allows us to seamlessly move between asynchronous editing approaches (which require copied view versions) and synchronous editing, which assumes the "same" data is being edited. In our approach, users who are synchronously editing a view actually still

have their own version of the view data, but these replicated versions are kept synchronised by automatically applying all changes made by other users to them.

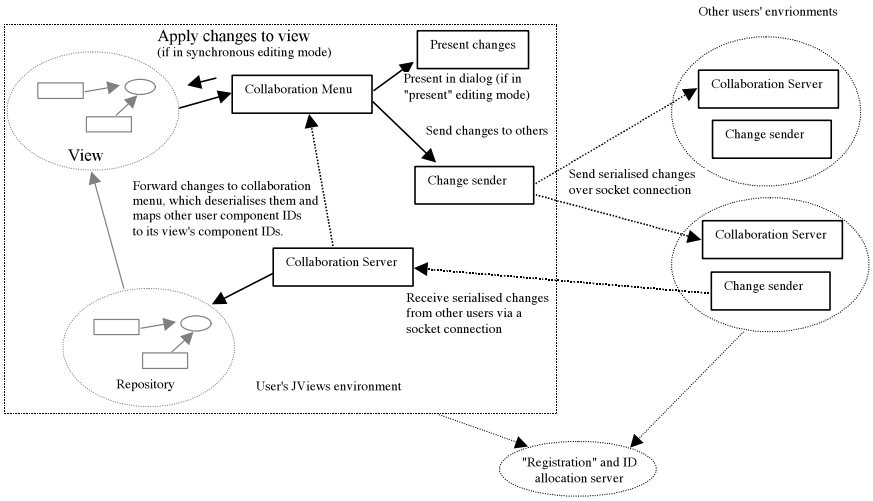


Figure 4. Communication of events between user environments

4. USER INTERFACE COMPONENTS FOR COLLABORATION

This section illustrates the human-interface aspects of our collaborative editing components, using JComposer as an example environment. We also relate the user interface characteristics to the various collaboration components described in the previous section.

4.1 Configuring Collaboration

When a JViews-based environment is first started, a saved JViews component must be opened by the user. When opened this automatically instructs the environment to add a collaboration menu to every new view, and initialises a single collaboration server for the environment. It also registers the user's name and host/port with the collaboration registrar server.

Whenever a JComposer view is created, a collaboration menu component is created and attached to this view. This component adds a "Collaboration" menu item to the view's pull-down menu bar, as illustrated in Figure 5. This menu allows the user of the JComposer environment to add collaborators, change the level of collaboration with specific collaborators, and to exchange the view components or change descriptions with other users (for asynchronous work). We chose this simple, menu-based interface for simplicity and to allow users to configure collaborative editing from one place.

When the user selects the "Add Collaborator" item, the collaboration menu component queries the collaboration registrar server for all other users names and host/port numbers. A selected collaborator is added and the collaborative editing mode with this collaborator set to "asynchronous" (level 1). Multiple collaborating users can be specified in this manner for the same view, and collaborative editing can be undertaken at different levels for each collaborator. The user changes the collaboration level using the Current Collaborators item and the collaborator's name subitems, as shown in Figure 5. We have found this approach allows for quick, seamless transition between levels of collaboration.

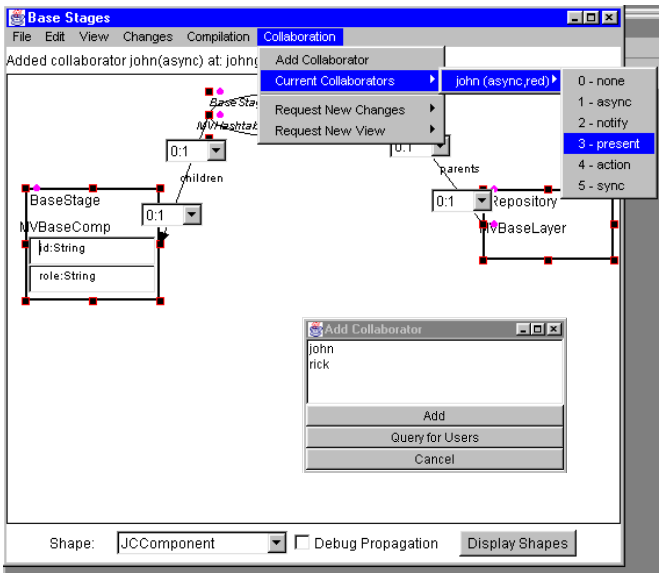


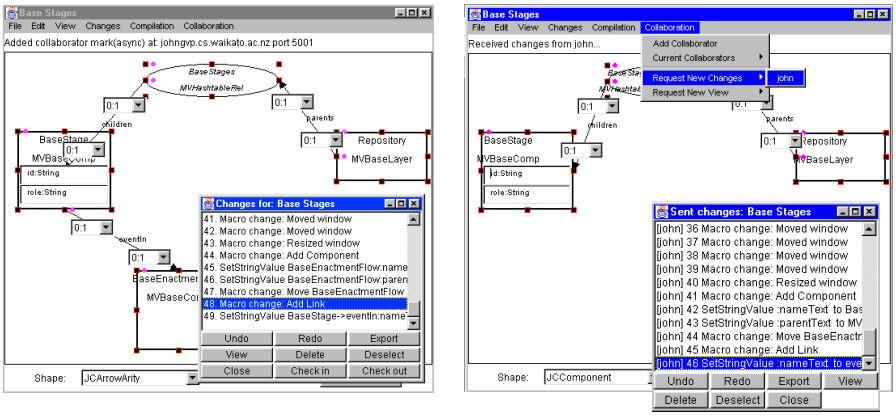
Figure 5. Specifying the collaboration "level" with other users for a view

When the collaboration level with another user is changed, a message is sent to the other user's collaboration server which changes their collaboration level with this user to match the one specified. We inform the user with whom the collaboration level has changed of the new level with a short text message displayed below the pull-down menus.

When a user first initiates collaborative editing of a view with another user, the collaboration menu component checks to see if the other user in fact has a copy of the view. If not, the view components are serialised and sent to the other user's environment. The components are then deserialised, given new unique ID numbers and the copy of the view displayed. The original ID numbers of the copied components are stored and used to map component ID numbers between different users' environments. Users always have a copy (i.e. alternate version) of a view they are collaboratively editing, resulting in fast editing response (no database or server-based data needs updating). This also allows users to move from asynchronous to synchronous editing seamlessly.

4.2 Asynchronous Editing

When users are editing a view with asynchronous collaboration ("level 1"), no changes to the view made by either users are propagated to the other user's environment. Instead, changes are only exchanged using the "Send Changes" menu item. Users can also choose to send a whole view definition to a collaborator, if preferred. When "Send changes" is used, as shown in Figure 7, all changes made to the view by the user since the last sending of changes are sent to the specified other user. These changes are then presented in a version record dialogue box, and the other user can selective merge them into their version of the view by selecting changes and clicking the "redo" button. At present we do not allow users to request changes from other users via a menu option, but rather leave the sending of changes under the control of the person who made them. This could be easily modified, if desired, but we feel users should communicate e.g. using audio or messaging, to ensure changes sent as required.

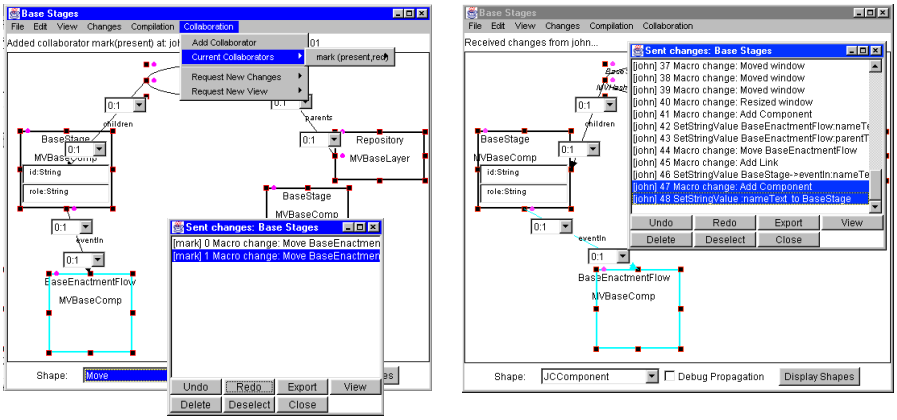


(a) John's view. (b) Mark's view.

Figure 6. Asynchronous editing of a JComposer view

Components and change descriptions are sent to other users by the collaboration menu component by serialising them, sending them to the specified user's collaboration server, and then having the receiving user's view deserialise them and map the originating view's component IDs to the receiving view's component IDs. Changes that can not be applied by the receiver e.g. they have deleted a view component the other user has edited, are marked as "invalid".

4.3 Presentation



(a) John's view. (b) Mark's view.

Figure 7. Presenting changes as they are made by other users

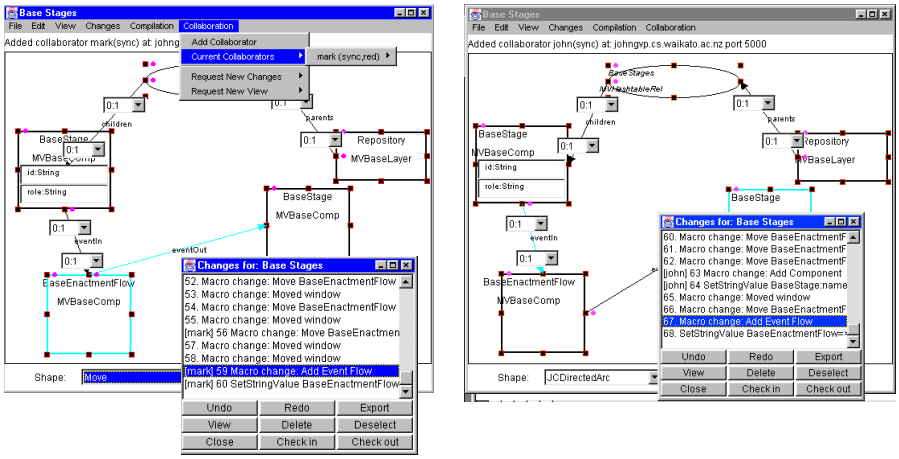
In the presentation level of collaborative editing ("level 3"), changes are broadcast as they are made to collaborating users, but then are presented in a dialogue box, as illustrated in Figure 8. Users then select the changes they desire to be made to their version of the view, and clicking on the "redo" button applies these changes to their view. Changes are tagged with the name of the user who originally made the change and are actioned by a receiving user. We have found this mode of collaborative editing very useful when users wish to keep informed of changes being made by other users to a shared view, but don't want these changes automatically performed on their version of the view.

4.4 Synchronous Editing

We provide two levels of "synchronous" - actioning ("level 4") and fully synchronous ("level 5"). Actioning changes simply involves the collaborative menu component applying received changes immediately to the view, rather than presenting them in a dialogue like level 3 presentation of changes. No locking or total ordering of changes protocols are used so users can potentially simultaneously edit view items, resulting in one change being immediately superseded by another when they are propagated to collaborators' environments.

Level 5 (fully synchronous) editing is the same as actioning changes but a locking protocol is employed, ensuring simultaneous edits do not occur. When a user begins to make an editing change, the collaboration menu component detects this (it is listening before and after state changes in the view occur, so gets a "before change" event from the view), and tries to "lock" the view component being changed. This is done by broadcasting a lock message to all other collaborators in level 5 collaboration mode on this view and waiting for a reply from each. Simultaneous attempts to edit the same view component result in neither user being able to obtain a lock, and we colour the view item red to indicate this. The users then

must wait and try and make their change again. Figure 9 illustrates synchronous editing in use, with the history of changes made by John and Mark shown in dialogues.



(a) John's view. (b) Mark's view.

Figure 8. Example of synchronously editing JComposer views

4.5 Mixed-Levels of Collaborative Editing

A user, "John", can simultaneously be at e.g. presentation level of collaborative editing with another user, "Mark", and at e.g. the synchronous level with a third user, "Steve". When synchronous changes between John and Steve are processed, Mark has the change presented to him. When Mark edits the view, John has the change presented to him, and when he asks for these changes to be actioned, Steve sees the changes made synchronously. At any time the collaboration levels between these three users can be changed, a user can end collaboratively editing their version of the view, or a fourth user can begin collaboration at any level.

JViews view inconsistency management techniques are used to ensure views are kept consistent or that users are aware of inconsistencies (Grundy et al., 1996). JViews allows users to incrementally merge asynchronous and presented changes with their views, and can detect if a change can not be actioned e.g. the user deleted the view item the change refers to, the user has not yet actioned a create view item change when trying to action a subsequent change etc. This allows users to very flexibly choose when changes made by others are actioned for their views, and tolerates view inconsistency for a period of time, which we have found very useful. Our environments can be configured to highlight and/or query for those view items which have unactioned changes associated with them, allowing users to monitor such inconsistencies and to later resolve them.

5. DISCUSSION

Many tools support asynchronous editing, such as Mjølner software development editors (Magnusson et al., 1993), the BSCW document management tool (Bently et al, 1995), Lotus Notes (Lotus, 1993), and editors like emacs. Other tools support synchronous collaborative work, such as GroupKit (Roseman and Greenberg, 1996), CoolTalk (Netscape Communications, 1995), and NetMeeting (Microsoft, 1997a). Some of these systems, such as NetMeeting and Mjølner, support aspects of both synchronous and asynchronous editing, but do not allow users to readily move between these modes of collaboration for the same view.

Other examples of combining different modes of collaborative work include SPADE/ImagineDesk (Di Nitto and Fuggetta, 1995), wOrlds (Bogia and Kaplan, 1995), Oz (Ben Shaul and Kaiser, 1997), TeamRooms (Roseman and Greenberg, 1997), and W4 (Gianoutsos and Grundy, 1996). These approaches generally separate the synchronous editors from the asynchronous ones, with little ability to seamlessly move between modes of operation. Additionally all of these systems have had these facilities built in from scratch. To add such capabilities to single-user tools usually requires major tool rearchitecturing and reimplementation.

Some component-based approaches to supporting collaborative editing include Orbit (Kaplan et al, 1997), CoCoDoc (ter Hofte et al, 1997; ter Hofte, 1998) and Emmerich (Emmerich, 1996), all which use CORBA-based object management and change propagation mechanisms. CoCoDoc supports collaborative compound document editing with multiple coupling levels, that can be changed independently for each component in a compound document. However, it does not allow users to have different levels for different pairs of users and does not support one user to have multiple views of a compound document open simultaneously. Other systems generally do not allow users to change collaboration levels on the same views, nor can these collaboration facilities be plugged into single-user environments in the manner we have provided.

Suite supports flexible coupling of user interface components (Dewan and Choudhary, 1991), allowing users to move between tightly coupled (synchronous) and loosely coupled (asynchronous) editing, with a variety of intermediate strategies. While the Suite approach to specifying coupling levels is similar to ours, and levels of coupling have some correspondance, we have found from our experiences with JComposer that setting coupling levels for whole views is sufficient. We have also found the use of change objects to be more flexible and provide a viewable, interactable history of work, in contrast to the Suite approach of less-flexible database-style querying to synchronise values. Our plug-in approach to adding collaborative editing is quite different to the attribute-based scheme of Suite, and we believe more reusable in general as tools continue to move to component-based software architectures. COAST (Schuckman et al., 1996) provides a component-based approach to building primarily synchronous collaborative editing systems. This utilises an MVC-style architecture, similar to that of JViews, but has much more limited view consistency management strategies and asynchronous support. Our use of the JViews, and hence Java Beans, event mechanism to support collaboration is different to COAST which provides its own message passing infrastructure. The JavaBeans composition tool of (Banavar et al., 1998) allows end users to compose their own collaborative editing applications using direct manipulation. However, these composed editors do not provide the degree of

flexible coupling of our JComposer tool, nor does this architecture allow users to add collaborative editing facilities directly to any JavaBean.

In order to use our approach, however, does require tools to utilise a component-based architecture, and an architecture with the flexible nature like JViews. Listening both before and after changes have been carried out, and even before changes sent to a listened component have been handled by it, is necessary to support both fully synchronous editing and to be able to multithread sending changes, receiving changes and user edits without interference. Unfortunately conventional component-based architectures like COM/DCOM (Microsoft, 1997b), JavaBeans (Javasoft, 1997), and that of TeamRooms (Roseman and Greenberg, 1997b), do not directly support such flexible component interconnection and event subscription. This makes reuse of our components with tools built with such architectures difficult, without substantial change of the tools to ensure appropriate event generation and propagation.

Collaborative editing components have aspects which involve human computer interaction e.g. the collaboration menu for configuring collaboration, the dialogue box to present changes in, and the highlighting of icons to indicate changes made by other users. There are limitations to what degree of seamless awareness and interaction can be provided using a component-based approach without modifying existing tool implementations or making collaboration components overly-dependant on particular tools. For example, generally the way icons are drawn can not be changed, and only certain kinds of highlighting of view items can be achieved. This limited the degree of interaction and awareness capabilities we provided when building our collaboration components for JViews-based environments. Aspects that involve the management of distributed objects, the propagation of change notifications between environments, and the handling of received changes are also important. We found it relatively straightforward to build these capabilities using JViews, although our use of unique IDs to tag JViews components is simplistic. We did not encounter any performance problems when using a component-based approach, compared to building similar collaborative editing facilities for a predecessor of JViews using conventional programming techniques (Grundy et al., 1995).

A final observation is the need for additional communication techniques to complement collaborative editing facilities. We found email-like messaging useful in conjunction with asynchronous editing, and an audio link useful with synchronous editing. An audio link and/or textual synchronous chat is useful with presentation level of collaborative editing, allowing users to discuss changes.

6. SUMMARY

We have described a component-based approach to adding user-configurable collaborative editing facilities to existing component-based design tools. This approach has numerous advantages over conventional approaches of building in collaboration facilities into editing tools. Users can move between different levels of collaborative editing for any diagram; components supporting collaborative editing can be plugged into any component-based tool implemented with our JViews architecture with no changes to the tool or collaborative editing components needed,

and these components can be easily replaced with new versions which provide better performance or facilities.

We are currently building additional collaborative work supporting capabilities into our collaboration components, including semantic telepointers for use in synchronous editing mode which show other users' mouse movements and menu interactions. We are investigating the use of CORBA or DCOM object persistency mechanisms to replace the simple object serialisation and persistency of JViews, and to provide an improved approach to object identification. We are also developing formal specifications of our collaborative editing components using Object-Z, to more formally specify their interfaces, behaviour and the kinds of tools into which they can be plugged.

ACKNOWLEDGEMENTS

The many helpful comments of the anonymous reviewers are gratefully acknowledged, as are the efforts of Henri ter Hofte, who's freely offered advice, comments and editing of earlier versions of this work go well above and beyond the call of duty. This work was supported in part by a grant from the Public Good Science Fund.

REFERENCES

Banavar, G., Miller, Doddapaneni, S., Miller, K., and Mukherjee, B. 1998. Rapidly Building Synchronous Collaborative Applications By Direct Manipulation*, Proceedings of the 1998 ACM Conference on Computer-Supported Cooperative Work*, ACM Press.

Ben-Shaul, I.Z., Heineman, G.T., Popovich, S.S., Skopp, P.D. amd Tong, A.Z., and Valetto, G. 1994. Integrating Groupware and Process Technologies in the Oz Environment, *Proceedings of the 9th International Software Process Workshop,* IEEE CS Press, Airlie, VA, October, pp. 114-116.

Bentley, R., Horstmann, T., Sikkel, K., and Trevor, J. 1995. Supporting collaborative information sharing with the World-Wide Web: The BSCW Shared Workspace system, *Proceedings of the 4th International WWW Conference,* Boston, MA, December.

Bogia, D.P. and Kaplan, S.M. 1995. Flexibility and Control for Dynamic Workflows in the wOrlds Environment, *Proceedings of the Conference on Organisational Computing Systems,* ACM Press, Milpitas, CA, November.

Dewan, P. and Choudhary, R. 1991. Flexible user interface coupling in collaborative systems, *Proceedings of ACM CHI'91*, ACM Press, April 1991, pp. 41-49.

Di Nitto, E. and Fuggetta, A. 1995. Integrating process technology and CSCW, *Proceedings of IV European Workshop on Software Process Technology,* Lecture Notes in Computer Science, Springer-Verlag, Leiden, The Netherlands, April.

Emmerich, W. 1996. An Architecture for Viewpoint Environments Based on OMG/CORBA, *Proceedings of 1996 International Workshop on Multiple Perspectives in Software Development,* ACM Press, San Francisco.

Gianoutsos, S. and Grundy, J. 1996. Collaborative work with the World Wide Web: adding CSCW support to a Web browser, *Proceedings of Oz-CSCW'96,* University of Queensland, Brisbane, Australia, August, pp. 14-21.

Grundy, J.C., Hosking, J.G., Mugridge, W.B., Amor, R.W. 199.5 Support for Collaborative, Integrated Software Development, *Proceedings of the 7th Conference on Software Engineering Environments*, IEEE CS Press, Noordwijkerhout, the Netherlands, April.

Grundy, J.C., Hosking, J.G., Mugridge, W.B. 1996. Supporting flexible consistency management via discrete change description propagation, *Software - Practice & Experience*, **20** (9), September, 1053-1083.

Grundy, J.C., Mugridge, W.B., and Hosking, J.G. 1997. A Java-based toolkit for the construction of multi-view editing systems, *Proceedings of the Second Component Users Conference,* SIGS Books/CUP, Munich, Germany, July 14-18.

Grundy, J.C. and Hosking, J.G. 1998. Serendipity: integrated environment support for process modelling, enactment and work coordination, *Automated Software Engineering*, **5** (1), January, 26-60.

Grundy, J.C., Hosking, J.G., Mugridge, W.B. 1998. Experiences in using Java on a software tool integration project, *Proceedings of 1998 Software Engineering: Education and Practice Conference*, IEEE CS Press, Dunedin, New Zealand, January 22-26 (in press).

Javasoft, 1997. *The Java Beans 1.0 API Specification*, Sun Microsystems Inc., See: http://www.javasoft.com/beans.

Kaplan, S., Mansfield, T., Phelps, T, Fitzpatrick, M., Qui, W., Taylor, R., Fitzpatrick, G., Berry, A. 1997. Orbit - supporting social worlds, *Proceedings of INTERACT97*, Chapman-Hall, Sydney, Australia, 114-116.

Lotus Corporation, 1993. *Lotus Notes Release 3*, See: http://www.lotus.com/.

Magnusson, B., Asklund, U., and Minör, S. 1993. Fine-grained Revision Control for Collaborative Software Development , *Proceedings of the 1993 ACM SIGSOFT Conference on Foundations of Software Engineering,* Los Angeles, December, pp. 7-10.

Microsoft, 1998a. *Microsoft NetMeeting 2.1*, Microsoft Corporation, See: http://www.microsoft.com/netmeeting/.

Microsoft, 1998b. *Component Object Model*, Microsoft Corporation See: http://www.microsoft.com/com/.

Netscape, 1996. *CoolTalk for NETSCAPE NAVIGATOR 3.0 BETA*, Netscape Communications, See: http://home.netscape.com/.

Roseman, M. and Greenberg, S. 1996. Building Real Time Groupware with GroupKit, A Groupware Toolkit, *ACM Transactions on Computer-Human Interaction*, **3** 1, 1-37.

Roseman, M. and Greenberg, S. 1997a. A Tour of Teamrooms, *Video Proceedings of ACM SIGCHI'97,* ACM Press, Atlanta, Georgia, March.

Roseman, M. and Greenberg, S. 1997b. Simplifying Component Development in an Integrated Groupware Environment, *Proceedings of the ACM UIST'97 Conference*.

Shuckman, C., Kirchner, L., Schummer, J. and Haake, J.M. 1996. Designing object-oriented synchronous groupware with COAST, *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, ACM Press, November 1996, pp. 21-29.

ter Hofte, G.H. and H.J. van der Lugt, CoCoDoc : A framework for collaborative compound document editing based on OpenDoc and CORBA. In J. Rolia, J. Slonim and J. Botsford eds, *Open distributed processing and distributed platforms : Proceedings of the IFIP/IEEE international conference on open distributed processing and distributed platforms*, Toronto, Canada, May 26-30, 1997. Chapman & Hall, 1997, pp. 15-33.

ter Hofte, G.H., *Working apart together : Foundations for component groupware. Telematica Instituut Fundamental Research Series, vol. 001*. Telematica Instituut, Enschede, the Netherlands, 1998, See: http://www.telin.nl/publicaties/1998/wat/wat.htm.

BIOGRAPHY

John Grundy is a Senior Lecturer in Computer Science at the University of Waikato, New Zealand. He holds the BSc(Hons), MSc and PhD degrees, all in Computer Science from the University of Auckland. His research interests include software engineering environments, software architectures and component-based software development, groupware systems, human-computer interaction and user interface technology, and object-oriented systems.