Timewarp: Techniques for Autonomous Collaboration

W. Keith Edwards and Elizabeth D. Mynatt
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
+1-415-812-4405
{kedwards,mynatt}@parc.xerox.com

ABSTRACT

This paper presents a set of techniques for supporting autonomous collaboration—collaboration where participants work independently for periods, and then join together to integrate their efforts. This paper posits that autonomous collaboration can be well-supported by systems in which the notion of time is made both explicit and editable, so that the parallel but divergent states of a shared artifact are exposed in the interface. We have developed a system, called timewarp, that explores these ideas, and provides support for distribution, awareness, and conflict resolution in an application-independent fashion.

KEYWORDS: Autonomous collaboration, computer-supported cooperative work, awareness, conflict detection and resolution, timewarp.

INTRODUCTION

The computer-supported cooperative work community has often classified collaborative sessions, and the applications used to support them, into two broad categories. The term *synchronous* collaboration is used to indicate that users are engaged in a tightly-coupled, same-time effort. Typically such systems present very fine-grained exchanges of information among users, and may approach a what-you-see-is-what-I-see (WYSIWIS) style of interaction.

The other broad category of interaction is *asynchronous* collaboration. While more loosely defined, asynchronous collaboration has typically been taken to mean collaboration that happens (or can happen) at different times. Group calendars and bulletin boards are the oft-cited examples. Users interact with some shared artifact, and this interaction doesn't *necessarily* have to happen at the same time. Even if it does happen at the same time, users may not be notified of the interactions of others since updates among users are not as fine-grained as in synchronous interactions.

Autonomous Collaboration

Recently, other styles of collaboration (perhaps not completely orthogonal to the ones mentioned above) have been named. One of these is *autonomous* collaboration [6]. While the term may seem to be an oxymoron, the notion of autonomous collaboration captures many of the styles of work seen in the everyday world. Autonomous collaboration is characterized by periods in which groups of users work *independently* on a *loosely-shared artifact*. These users then come together for periods of *tightly-coupled sharing* to *integrate* the disparate work done by collaborators. Coordination and comprehension of parallel, independent efforts necessitates *awareness of current and past efforts* among users.

Perhaps the most common example of autonomous collaboration is "traditional" (non-computer mediated) paper writing. Typically each writer will take a section of the paper, return to an office, and begin writing. Periodically, the writers will "synch up" to integrate their changes, and make sure their text is not contradictory or redundant.

Autonomous forms of collaboration are appealing because they loosen the constraints that more tightly-coupled forms of collaboration often impose. Collaborators are free to work wherever and whenever is convenient for them. Autonomous collaboration is also useful in situations where connectivity to the network is not always assured. Examples include mobile or at-home usage, or settings with unreliable network infrastructure.

Another common example of autonomous collaboration is group paper reviewing. Typically, an author will print several copies of a paper draft and give these to coworkers for review. Each coworker will—separately—annotate the paper and suggest revisions. The editor then takes these comments and integrates them.

These comments refer back to an instance of the paper as it appeared at a certain moment in time. As the editor integrates the suggested revisions, the document changes. While it shares some similarities with the document the reviewers commented on by virtue of its common history, it has also accreted a number of differences with that version. As more reviews are returned, the author is tasked with integrating changes that refer back to a version of the document that no longer exists. The editor needs the ability to edit the document as it existed when it was last shared between all of the authors irrespective of the modifications being introduced.

CHI 97, Atlanta GA USA

In this collaborative writing scenario, the editor also needs to be able to review how the document has changed over time as a result of the individual contributions by the authors. For example, the editor may need to survey changes made to a document during the past week, or to assess the contributions of a certain author. One author may need to determine what changes have made since he or she last visited the document. These situations show the importance of awareness in the setting of autonomous collaboration.

Time As A First Class Citizen

The central premise of this paper is that autonomous collaboration can be supported by making the notion of time explicit in the interface. One manifestation of this principle is that applications can provide multiple *parallel timelines* to organize and coordinate the work of independent collaborators. This is in contrast to most synchronous and asynchronous collaborations where the actions of users create a serial ordering, and conflicts are quickly detected or resolved.

As the paper reviewing example shows, when collaborators are only loosely sharing an artifact, each will have their own view of the state of the artifact, which will evolve more or less independently. By making time explicit, tools for autonomous collaboration acknowledge that people often work in parallel, but independently, for long periods of time, producing interim results that may conflict with the efforts of others. The users' style of work is represented in this model of collaboration.

Another aspect of making time explicit in autonomous collaboration is that time should be *malleable*. Once time is made visible to users, they should be allowed to interact with it to accomplish their goals. Put succinctly, the application's notion of time and history need not rigorously reflect these concepts in the "real world." Users should be able to modify and interact with document histories as needed.

For example, an author working on formatting for a large document project can modify the document's state at a point in the history logically *before* other collaborators split off to work independently on parallel versions. Changes made "prior" to the split are immediately reflected in the later versions. The ability to edit timelines is crucial to integrating the parallel work of collaborators.

Introducing Timewarp

We have developed an application toolkit, and a number of applications on top of this toolkit, to support autonomous forms of collaboration. The chief technique supported by these systems is the use of *multiple independent histories or timelines* of the shared state during the collaboration. Users can easily interact with documents and artifacts at arbitrary states of their development, and merge multiple timelines to produce a coherent, unified result. We call this technique and the toolkit that implements it *timewarp*.

Rather than treating a document or other artifact as a static entity that has a fixed value for any given point in time, timewarp applications make the document's entire history explicit in the interface. Users interact with the document independently at different points in its history, or they can work in different (but related) parallel histories.

Autonomous collaboration requires users to deal with multiple versions of a document, be able to notice and resolve conflicts among versions, and maintain awareness about the actions of others. Timewarp supports these activities by making the notion of time both explicit and malleable in the interface. So rather than requiring users to maintain information about versions and conflicts in their heads, we provide an external representation of document timelines that serves to mediate and coordinate the participants in a collaboration.

In this paper, we explore how the timewarp system meets the needs of autonomous collaboration. In the next section, we describe the basic timewarp system, and the timewarp model for presenting multiple parallel document timelines to collaborators. Since this work was inspired by efforts in programming-by-demonstration, collaborative drawing, and version control systems, we discuss how timewarp expands on these concepts. Next, we present a sample application built using the timewarp toolkit. We then address a number of issues that arise when history itself can be rewritten by collaborators: what are the ways in which users can profitably interact with time, how can we support both loose- and tight-coupling through these interactions, how can we deal with conflicts in a setting where users have divergent views of the state of the world, and how can awareness be supported in an autonomous setting. We close with a brief discussion of implementation details and a set of directions for future research.

TIMEWARP BASICS

As stated, the central notion in the timewarp toolkit is that, rather than forcing users to implicitly maintain a model of the multiple states that a shared document may be in during a loosely-coupled collaboration, we make the document timelines explicit in the interface.

But what does it mean to make timelines explicit? Timewarp applications typically present two views onto a document. The first is a window showing the state of the document at a *single* point in time. The second view shows *all* of the document's states, organized into histories called *timelines*. The sum of all timelines of the document constitutes the complete "universe" of the document—a meta-history of all timelines.

The viewer for this meta-history typically presents the timelines as a singly-rooted directed acyclic graph (see Figure 1). Each node in the graph represents a discrete state in the document's history. Each edge represents some action that was performed on the document by a user to advance it to a new state. By selecting nodes, users can jump to pre-existing states in the document. The system maintains a path through the graph, from the root, through the currently selected node, and out to some trailing node. By selecting a node outside their current timeline, users can jump to other parallel timelines.

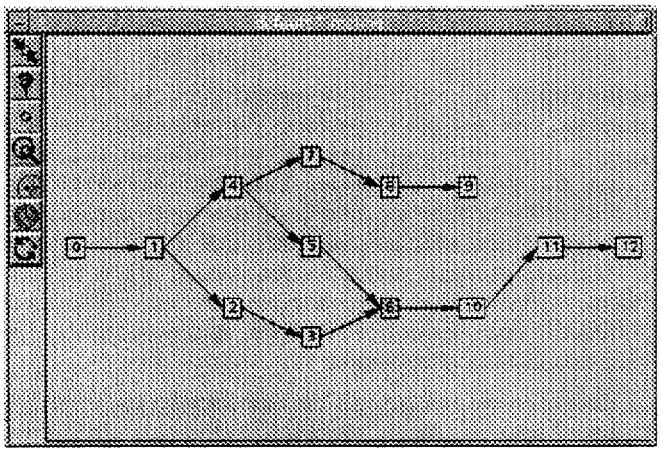

FIGURE 1: History View Window

The document viewer shows the shared document as it existed at the selected state, and provides the basic interface for editing and interacting with the document. A slider on the document viewer moves the current state through the current timeline. Users can go to "upstream" nodes—states that existed before the current one in the document's history—by dragging the scrollbar backwards. Likewise, users can visit "downstream" nodes—states after the current one—by moving the scrollbar forward. The document presented in the viewer changes as the scrollbar moves to reflect its state at the desired point in time.

Further, users can edit *any* state of a document, even if it exists upstream in a timeline. Changes can be made to propagate downstream from the point of edit to the present state of the document.

Figure 2 shows the document viewer window for one sample timewarp application, an office interior layout system.

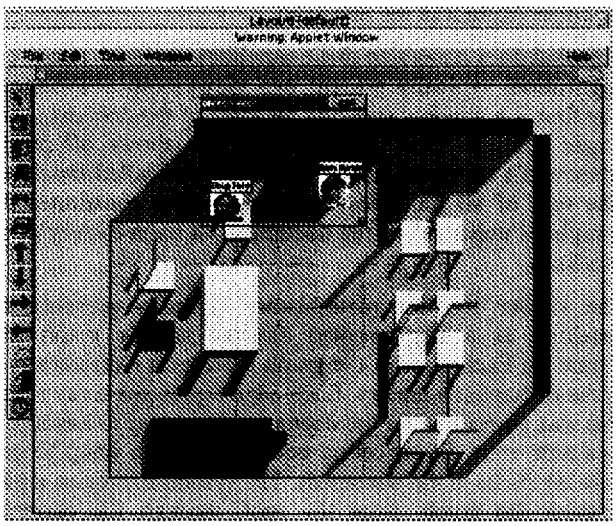

FIGURE 2: An Office Layout Application

While timewarp applications can be used in a single-user, stand-alone mode to organize versions and provide effectively infinite undo, these applications are most compelling when used by multiple users.

Timewarp allows multiple, distributed users to interact with a document autonomously. That is, any number of users can edit a document collaboratively. They may interact with the document at the same time or different times, but the system presents each with the complete global set of states that represents the document's meta-history. Users can explicitly share state with one another by working from portions of the history graph that share a common path. Alternatively, users can work on divergent paths of the graph, and integrate their changes later when convenient. As we will explain later, the system can detect potential conflicts, and allow users to resolve them.

To support coordination among users, timewarp provides a set of tools for enhancing awareness of users, actions, the state of the document, and its history. Users can fluidly traverse document timelines or jump to any point in a timeline. A set of "magic lenses" allows users to gather detailed information about the document state. The facilities for awareness are discussed in detail later in this paper.

Previous Work

To a large extent, the approaches presented in this work are inspired by work by Kurlander on Chimera [7], and by Rhyne and Wolf on WeMet [8]. Kurlander's system supports the construction and use of "editable graphical histories"—graphical representations of past actions in an editor that are themselves editable. Chimera raises the question of what are the implications of being able to make changes to the representation of a document at an earlier point in time.

Rhyne and Wolf present a collaborative handwriting and drawing surface. The system provides a scrollbar at the top of the window to move forwards and backwards in time. The system accommodates late joiners to a collaborative session by allowing them to replay the events that occurred prior to their arrival. The technique of traveling through a document's history is shown in a compelling way by this application.

Our work differs from these previous efforts in a number of ways. Chimera's primary focus is on the creation of by-example macros, not collaboration. While the system supports the editing of prior graphical states, there is only one timeline for the document, and upstream changes are always propagated downstream. Further, conflict detection and resolution do not appear to be addressed.

WeMet focuses on breaking down the barriers between synchronous and asynchronous collaboration. Like timewarp, it supports multiple users, each with multiple views of the document space, and allows branching of the timeline. Unlike timewarp, WeMet only supports simple divergence of the document timelines (similar to "split mode" in our system, see below). Further, there are no facilities for integrating "upstream" changes in the timeline, and thus no facilities for conflict detection and resolution.

There are no mechanisms for joining disparate timelines together, and no "meta view" that presents the entire set of parallel document timelines.

Other closely related research includes work by Berlage and Genau on Gina [2]. Like timewarp, the Gina work uses a toolkit approach, and a centralized server-based implementation. Gina provides special attention to merging using the technique of selective redo. Timewarp provides more flexible support for conflict handling, however, including the ability to tolerate outstanding conflicts caused by application semantics. Timewarp also provides more facilities for interacting with and editing the parallel timelines, and provides support for awareness among participants and across time.

Timewarp has some similarities with version control (VC) applications—both systems manage the progression of a document through time, by checkpointing the document at different versions, and notifying users of conflicts.

But there are a number of important differences between timewarp and most version control systems:

- Whereas in the VC model, differences are computed and conflicts are detected only when a check-in is performed, timewarp allows applications to define their own semantics for detecting differences and conflicts. Some applications may choose a continuous, "real time" computation of state changes, while others may opt for a more heavy-weight, coarser-grained approach.
- Timewarp provides a number of collaborative features for awareness, synchronous interaction, and replay that are not found in version control systems.
- Perhaps most importantly, VC systems do not allow upstream modifications to a document to propagate to the later downstream versions. If a revision is based on an upstream version, a split is created in the version tree. Timewarp supports a number of in-place editing modes that allow changes to propagate downstream.

Like timewarp, the Bayou system has support for application-defined conflict handling semantics [9]. But Bayou is primarily a data storage facility, without higher-level programming interfaces specific to collaboration. Bayou does not have the notion of explicit timelines, and does not have support for awareness.

AN EXAMPLE APPLICATION

We have built several applications using the timewarp toolkit, including a structured drawing editor and a text editor. Here we present an interior design application that allows users to collaboratively edit the layout of furniture and interior walls in an office setting.

The main window of the layout application is shown in Figure 2. The interface displays an office layout in an axonometric view[1] that can be scaled and scrolled. A palette on the left of the window provides tools for selecting, placing, rotating, and moving pieces of office furniture and interior walls. The scrollbar at the top of the application allows users to "move" through their current timeline. The menu at the top provides access to several other windows, including the "meta-history" view shown in Figure 1, a list of users (both current and inactive), and other tools.

One semantic constraint that the application places on users is that two objects cannot be located in the same place at the same time. When positioning objects on the floor plan, "snapping" is used to prevent objects from being placed in the same space.

We refer back to this particular application in later examples.

EDITING TIME

A powerful feature of timewarp is that it allows users to interact with a document at any point in its history, and affect later states of the document. This is in contrast to version control systems, changes to an early version of a file represent a new branch off of the version tree. They do not propagate outward to all derived versions.

In timewarp, document histories are malleable. Thus, changes can be made to a document at any point in its history, and these changes can be made to "ripple" out to later states. Sometimes these changes can result in inconsistencies; how timewarp deals with these inconsistencies is discussed later.

This ability is crucial for autonomous collaboration because it allows users to integrate changes that affect all collaborators globally. For example, if a number of users are editing a document, one user may insert an upstream change into the document's timeline to make the change immediately visible to all collaborators currently working at downstream states. The in-place editing of prior document states allows the system to, when necessary, act as a tightly-coupled synchronous editor where changes are immediately broadcast to all parties. This behavior is in contrast to the situation where users work independently on their own branches of the document history and integrate them later.

The timewarp tools support a number of "time editing" modes that govern how new changes to the document are inserted into the history graph. These modes are:

- Split mode
- Insert mode
- Retained mode
- Coupled mode.
- Join mode

Split Mode

Split mode is the default time editing mode for most applications. In split mode, a change made while the user is at a trailing (childless) node in the history graph causes a new state to be appended, derived from the first. If the user is at an interior node—that is, a node with children—the

1. An axonometric view is a "false" 3D projection, where all parallel lines stay parallel in the 2D representation.

history graph forks at that point. The old set of children remain and are unaffected by the change, while a new child state and edge are created.

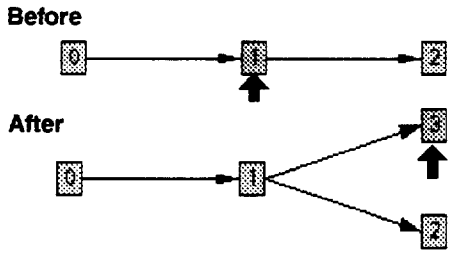

FIGURE 3: Effect of Split Mode

Figure 3 shows before and after pictures of a change to the graph in split mode. Before the change is made, node 1 is the current node. Afterwards, node 3 has been appended and made the current node.

Split mode is the principal tool that allows autonomous interaction in timewarp. Divergent timelines allow multiple users to interact with similar but disparate versions of a shared artifact. In essence, each user interacts with a private "parallel history" of the document, which may share some context in common with other branches.

Insert Mode

Insert mode allows in-place changes to be made to the document timeline. Insert mode handles changes made at trailing nodes the same as split mode: a new node is appended and the user continues. In interior nodes, however, insert mode causes a new state to be appended after the current node. The existing children of the current node are reparented to the new node.

This mode introduces a new change upstream in a timeline—in essence, the document's history is rewritten. All of the downstream states are effectively changed, as they now include the effect of the new action taken by the user.

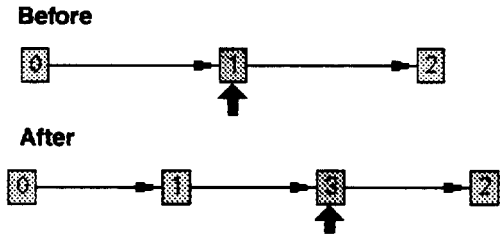

FIGURE 4: Effect of Insert Mode.

Figure 4 shows an insertion being done. Prior to the insertion the current node is 1. After the insertion, node 3 is created in-place and made the current node.

Insert mode has the potential to introduce a number of conflicts into the state of the system. See below for more discussion of conflicts. Insert mode is useful for making an immediate change that affects downstream nodes—and hence the users currently working at those nodes. It also can serve as an editing convenience in single-user use.

Retained Mode.

In insert mode, the original path that existed before the insertion is lost. Once the insertion is made there is no way to revert to the old history. This behavior is often desired if the change to be made is small or if the need to have the change is certain. At other times, though, users may wish to retain the original history when the insertion is made. Retained mode provides this behavior.

Retained mode works exactly like insertion mode except that when a modification is made to an interior node, retained mode *copies* the original downstream portion of the graph. A split is then made between the original downstream nodes and the new path that contains the inserted change. In a sense, retained mode is a "safe" version of insert mode that does not lose the original timeline.

Figure 5 shows an example of retained mode.

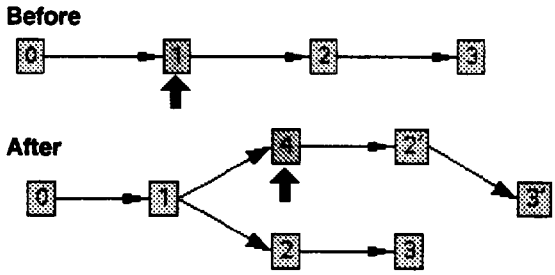

FIGURE 5: Effect of Retained Mode

Join Mode

The modes discussed so far allow users to split the document timeline, or add changes in place. For collaborative applications, however, the goal is typically to produce one result at the end of the collaborative task. Particularly, in the case of autonomous collaboration, having flexible tools that allow users to work along divergent paths is useful, but in most cases users still need to merge those divergent paths into one final state that represents "the finished product."

Join mode allows states on multiple paths to be joined together, producing a new state representing the union of the actions taken in the two parent paths. Like the modes that allow insertion along a timeline, join mode can result in conflicts if the set of actions taken along the two parents are not compatible with one another.

Figure 6 shows an example of join mode.

Join mode facilitates the stage of autonomous collaboration where users come together to merge a set of disparate changes into one result after working separately for a period.

DEALING WITH CONFLICTS

Much of the power of timewarp is that it allows a document to be edited at any point in its history. Changes can be inserted upstream in a timeline, which then propagate to all derived states of the document. However, allowing such interactions with past and parallel states introduces the potential for conflict.

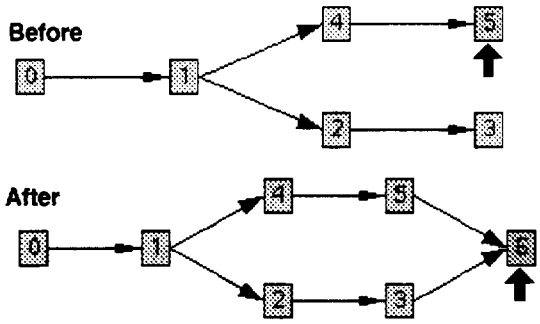

FIGURE 6: Effect of Join Mode

As an example, consider a sequence of operations in the layout application. A user may create an object, place it on screen, move it, copy it, and paste it any number of times. If the user goes to the point in time prior to when the object is copied and deletes it, a potential conflict exists: the object just deleted is referenced later in this timeline.

Making earlier states editable—a situation not commonly found in the real world where history typically goes unchanged—introduces the potential for a class of paradoxes[2] common in science fiction writing: you can go back in time and prevent yourself from being born, but then how can you go back in time in the first place?

Such operations are conflicts because they violate *temporal dependencies* in the history of the document. A temporal dependency indicates that an ordering relationship exists between two operations. In the example above, the object must exist if it is to be copied and pasted.

Temporal dependencies exist because of the editable nature of timewarp timelines. But other types of dependencies can exist because of application constraints as well. For example, the layout editor has a *spatial dependency*: no two objects can exist in the same place at the same time. The semantics of other applications may introduce new classes of dependencies, each of which can generate conflicts if the dependencies are violated.

A complete discussion of the mechanisms provided for conflict detection and resolution is outside the scope of this paper, although the Implementation section does give a high-level picture of these mechanisms. In short, the timewarp toolkit provides a set of application-independent facilities for dealing with conflicts that fall into three classes:

- **Detection**
  Applications can specify facilities for detecting domain-specific conflicts. These facilities are used to evaluate whether a conflict exists each time the timeline is modified.

2. In our system, we call paradoxes "conflicts." Saying that conflicts can arise during the execution of an application seems somehow less threatening to computer scientists than saying that an application causes paradoxes.

- **Visualization**
  Applications can provide their own visualizations for presenting conflicting data to the user. For example, the timeline scrollbar may be decorated with information showing the locations of conflicting states or, as in the case of the layout application, conflicting objects may be rendered differently.
- **Resolution**
  Applications may provide their own policies for resolving conflicts, complete with a user interface for interactions with users when help is needed to resolve a particular conflict. One example of a policy for dealing with temporal conflicts is to sever the dependency between conflicting objects. For example, an object resulting from a cut and paste operation may be instantiated as a "stand-alone" object, even if the originally cut object is deleted.

The timewarp facilities for handling conflict can tolerate ambiguity—that is, certain applications may be able to operate in the face of inconsistent data. For example, the layout application does not rigorously enforce spatial separation of objects. If an insertion or join operation results in two objects being at the same location (a violation of application semantics), the conflict system renders the overlapping objects transparently to indicate that they are in an ambiguous state (see Figure 7). Users can choose to tolerate this ambiguity, or resolve the conflict by editing the layout.

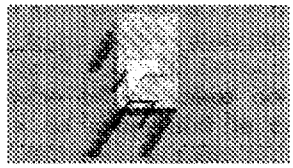

FIGURE 7: Two Overlapping Objects Are Drawn Transparently

SUPPORTING AWARENESS

Awareness is the trait of knowing about the environment you exist in: what are your surroundings, who is around you, what are they doing. Awareness is essential in collaborative activities because it gives us an indication of how to correlate and coordinate our activities with others [4].

Most work on awareness, however, has focused on what might be termed *synchronous awareness*—awareness about what is going on around you *right now*. In contrast, autonomous collaborations require a different form of awareness. Since the collaborators are only loosely-coupled—they may not be working at the same time, and they are almost certainly not working at the same state of the document—synchronous awareness techniques by themselves are insufficient.

We have explored a number of techniques for providing awareness in an autonomous setting in timewarp. These techniques provide a sense of the overall history of the

document, as well as specific, directed forms of awareness information. These are described below.

Awareness of History

The most immediate and important form of awareness is awareness about the history and state of the document being shared.

Timewarp provides two mechanisms for users to become aware of the history of the document they are interacting with. The first is the timeline scrollbar at the top of the document viewer. This scrollbar allows users to easily replay any actions that occurred at any time in the document's history. Changes in state are accompanied by animation that highlights the salient differences between states.

By interacting with the timeline scrollbar, users gain a sense of the relationships between document states, and the changes that a document has gone through.

The second mechanism is the global "meta-history" viewer. This window presents a "gestalt" view of the document's history. Whereas the scrollbar allows interaction along one particular path in a document's history—the currently selected one—the graph view allows users to see all the parallel timelines of a document. Users can see the "big picture" of the document's history, visit any state, and see what collaborators' views of the world are.

Magic Lenses for Autonomous Collaboration

In addition to techniques for examining the document's overall history, and the changes between states, timewarp also provides a set of tools for presenting very specific forms of awareness information to users. Most of these tools take the form of *magic lenses* [3]. Magic lenses (or just "lenses") are an interaction technique for selectively presenting alternative views of information on a two-dimensional surface. A lens is dragged over the surface, and the information viewed "through" the lens is altered in some way to add or filter information.

Timewarp provides a number of lenses that can be used by collaborators to inspect the document and the shared history. Figure 2 shows the "Identify User" lens being passed over a floor plan in the layout application.

Knowing About People. Perhaps the most important kind of specific awareness information is knowing about fellow collaborators. Timewarp provides a set of lenses for retrieving information about people in a collaboration.

- **Identify User.** The lens identifies the creator of any object it passes over.
- **Filter User.** Only objects created by a specified user are displayed. The rest are not shown by the lens.

Knowing About Actions. A second broad category of lenses presents information about the actions that have taken place in a document's history. These include:

- **Show Actions.** The edges on the graph view of history are annotated with description of the actions that took place at those transitions.
- **Show Conflicts.** The graph is decorated with information showing conflicting actions.

Knowing About Time. The third category of awareness lens shows information about time.

- **Difference States.** This lens graphically shows the differences between the current state and another state.
- **Summarize Change.** This lens shows regions of the document where the most activity has taken place. Regions of high activity are presented "smudged," as if a number of users had left fingerprints there.

Taken together, this set of lenses provides users with tools to selectively acquire specific information about other users, states, and activities in an autonomous collaboration.

IMPLEMENTATION

The timewarp toolkit and applications are implemented in the Java language [1], using the subArctic GUI toolkit [5]. Any timewarp application can be run as a stand-alone, non-collaborative tool, or can be used collaboratively among a group of users. The system uses the Java Remote Method Invocation (RMI) system for communication between Java Virtual Machines (JVMs, the actual Java bytecode interpreters) running on different host computers. The toolkit is roughly 15,000 lines of code.

Building New Timewarp Applications

The toolkit is designed to be completely domain-independent. To write a new timewarp application, a developer must do two things: (1) implement the user interface that allows interaction with the document or other artifact being shared. The various event handlers of this interface should be tied to toolkit code that alters the state of the history graph. (2) Implement a number of *action* classes that represent the allowable operations supported by the application that can affect the timeline of the document.

Actions are the atoms of behavior in a timewarp application. The set of actions for the layout program includes CreateObject, MoveObject, RotateObject, Cut, Copy, and Paste. The timewarp infrastructure requires that every action be reversible. Thus, there is a base Action class from which application writer-supplied actions are derived. This base class requires two methods called forward and reverse that can be used to run the action in either direction.

When the user makes a change to the shared document, a new Action instance representing that change is created. The timewarp toolkit creates a new edge and node in the history graph, and stores the action at the newly-created edge according to the current time editing mode. Thus, the action is placed in the graph to represent the change necessary to move between two adjacent states in the graph.

The timewarp infrastructure currently only stores the actions needed to reach states in the graph—the states themselves are not retained. For very large graphs or for applications where actions are complex, this strategy may become

problematic. For our applications, however, the cost to recompute state on-the-fly has not been excessive.

Distribution

The timeline graph and the actions stored in it are the only state that must be shared among a collection of cooperating applications. Thus, when a new timewarp session is started, a graph server process is started to manage interaction among clients who are editing a single document.

This server process is also implemented in Java and communicates with clients via RMI. The server supports late-joiners by transmitting required graph state when a new connection is initiated. The client-side stub code within the Timewarp toolkit implements some fairly extensive caching to provide good performance across slow links or in the case of large numbers of users. This caching machinery is not seen by application code.

Conflict Handling

The timewarp toolkit provides basic infrastructure for detecting and resolving conflicts. When a new edge is inserted into the history graph, a closure is applied to all downstream edges to see if a conflict results. Timewarp provides facilities to application writers to define their own closures to decide what constitutes a conflict for their particular application domain.

If a conflict is detected, it is passed to a conflict policy object that decides how to handle the conflict. Timewarp provides a number of conflict policy objects that implement a range of policies, and a number of user interfaces to conflict resolution. Application writers are free to provide their own conflict policy objects if the provided ones are insufficient.

The basic detection and resolution machinery is application-independent—application writers provide code that implement the semantics particular to their applications. In most cases, application writers will only need to supply a handful of conflict closure classes that will be used to detect conflicts, and perhaps a new conflict policy object to be used to dispose of detected conflicts.

STATUS AND FUTURE DIRECTIONS

So far, we have written three applications based on the timewarp toolkit: a structured graphical editor. a text editor, and the office layout program. All three of these systems share most code in common. Only the graphical interface for the document viewer and a handful of action classes were needed in each case to construct fully timewarp-aware collaborative applications. The office layout program also adds some application-specific conflict detection, visualization, and resolution code not provided by the timewarp base classes.

Architecturally, our experiences with the system have been positive. The facilities for defining and handling types of conflicts specific to application semantics are extremely important for autonomous application. Further, the underlying model, supporting loose sharing of an document by making the document's history explicit, is a useful technique for supporting autonomous collaboration.

There are a number of areas we plan to explore further, however. First, we plan to examine how to support more tightly-coupled synchronous collaboration in timewarp applications, for situations where users need fine-grained interaction. We are investigating more lenses for awareness, and we also intend to refine our ideas about conflict management and user interfaces for dealing with conflicts and ambiguity. In the longer term, we are interested in supporting even looser forms of collaboration by moving the timewarp infrastructure from a centralized server approach to a more fully distributed, weakly-consistent architecture.

REFERENCES

[1] Arnold, Ken, and Gosling, James. *The Java Programming Language*. Addison-Wesley Co., Reading, Mass. 1996.

[2] Berlage, Thomas, and Genau, Andreas. "A Framework for Shared Applications with a Replicated Architecture." In *Proceedings of ACM Symposium on User Interface Software and Technology* (UIST). November 3-5, 1993, pp. 249-257.

[3] Bier, Eric A., Stone, Maureen C., Pier, Ken, Buxton, William, and DeRose, Tony D. "Toolglass and Magic Lenses: The See-Through Interface." In *ACM Computer Graphics Proceedings*, August 1993, pp. 73-80.

[4] Dourish, Paul, and Bellotti, Victoria. "Awareness and Coordination in Shared Workspaces." *Proceedings of ACM Conference on Computer Supported Cooperative Work* (CSCW), November 1992. pp. 107-114.

[5] Hudson, Scott, and Smith, Ian. "Ultra-Lightweight Constraints." *Proceedings of ACM Symposium on User Interface Software and Technology* (UIST), 1996.

[6] Kolland, Markus. "Distributed System Support for Consistency of Shared Information in CSCW (Extended Abstract)." Workshop on Distributed Systems, Multimedia, and Infrastructure. ACM Conference on Computer-Supported Cooperative Work (CSCW), Oct. 22, 1994.

[7] Kurlander, David, and Feiner, Steve. "A History-Based Macro by Example System." In *Proceedings of ACM Symposium on User Interface Software and Technology* (UIST). November 15-18, 1992. pp. 99-106.

[8] Rhyne, James. R., and Wolf, Catherine G., "Tools for Supporting the Collaborative Process." In *Proceedings of ACM Symposium on User Interface Software and Technology* (UIST). November 15-18, 1992. pp. 161-168.

[9] Terry, Douglas B., Theimer, Marvin M., Petersen, Karin, Demers, Alan J., Spreitzer, Mike J., and Hauser, Carl. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System." Xerox PARC Technical Report CSL-95-4, August 1995.