A Temporal Model for Multi-Level Undo and Redo

W. Keith Edwards
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304
kedwards@parc.xerox.com

Takeo Igarashi[1]
Brown University
CS Dept., Box 1910
Providence, RI 02912
takeo@cs.brown.edu

Anthony LaMarca[1]
Yahoo, Inc.
3420 Central Expwy
Santa Clara, CA 95051
lamarca@yahoo-inc.com

Elizabeth D. Mynatt[1]
College of Computing
Georgia Tech
Atlanta, GA 30332
mynatt@cc.gatech.edu

ABSTRACT
A number of recent systems have provided rich facilities for manipulating the *timelines* of applications. Such timelines represent the history of an application's use in some session, and captures the effects of the user's interactions with that application. Applications can use timeline manipulation techniques prosaically as a way to provide undo and redo within an application context; more interestingly, they can use these same techniques to make an application's history directly manipulable in richer ways by users. This paper presents a number of extensions to current techniques for representing and managing application timelines. The first extension captures causal relationships in timelines via a nested transaction mechanism. This extension addresses a common problem in history-based applications, namely, how to represent application state as a set of atomic, incremental operations. The second extension presents a model for "multi-level" time, in which the histories of a set of inter-related artifacts can be represented by both "local" and "global" timelines. This extension allows the histories of related objects in an application to be manipulated independently from one another.

KEYWORDS: history management, timelines, undo, redo, Timewarp, Flatland.

INTRODUCTION
Many applications provide facilities for interacting with their past states. The most common expression of such facilities is as a way to support undo and redo of user operations. Applications as commonplace as Microsoft Word support such features, allowing users to effectively "roll back" the history of their interaction with the application.

More recently, a number of systems in the research literature have explored richer models of application history. These include such tools as GINA [1] and Amulet [13], which use a complex branching model of time for undo and redo operations; Chimera [12], which provides a history model which preserves dependency relationships between operations; WeMet [16], which provides tools for rapidly scanning shared timelines; and Timewarp [4], which uses divergent and convergent timelines as a way to support collaboration, and provides facilities for conflict resolution. Even in systems without such rich models of history, time is often an explicit—and directly manipulable—part of the user interface, and user experience. Systems such as Time-Machine Computing [15] and Lifestreams [6] are exemplars of this trend.

All of these systems rely on an explicit model of history, which can be scanned to support search or "navigation" over a timeline, and all allow their timelines to be "traversed" to move the application's state to other points in its history. However, as powerful as these applications are, their timeline representations are for the most part exceedingly simple. They typically support only linear, not branching timelines (GINA and Timewarp are exceptions, however); the "nodes" in a timeline must represent atomic operations with side effects that are well understood at the time the application is created; and, typically, the timeline of the entire application must be navigated or traversed as a whole—it is impossible to have a portion of the timeline exist in a "bubble" that can be manipulated separately.

While we don't commonly encounter such rich models of time in our day to day experience, they *can* be extremely useful nonetheless. Divergent timelines, for example, can be employed to allow users to interact with different but related versions of an artifact, and then reconcile those differences later. The ability to expand the representation of time in ways that better accommodate side effects can make applications easier to write. And being able to separate the history of one nested artifact from the history of the application as a whole can allow users to work locally on a document, project source code, et cetera, and still integrate their changes globally.

This paper presents an expansion of the most traditional representation of timelines, which is based on the command object idiom. The research here makes two contributions. First it extends this traditional model of history to better support the causality effects often found in "real" applications. This first extension works for both linear and divergent timelines. Second, this new causal model is then extended to support multi-level or "interleaved" timelines, in which the various components of an artifact can exist at

UIST '00. San Diego, CA USA

1. All authors were at Xerox PARC when this research was conducted.

different points in the overall global timeline. This second contribution only addresses interleaving in the context of linear timelines, not divergent timelines.

BACKGROUND: HISTORY MODELS FOR SINGLE-LEVEL UNDO/REDO

As mentioned, many desktop applications provide a general-purpose undo/redo mechanism. One common approach to implementing such a facility is based on the "command object" idiom [7]. Command objects are objects—in the object-oriented sense of the word—that encapsulate a change in the state of an application. Typically, each *type* of operation that can be performed in the application is represented as a discrete *class* of command object. Instances of these commands are "invoked" by calling a well-known method on them that causes them to perform their operation, updating the state of the application. In other words, instances of these command classes represent *particular* invocations of these operations by the user, and also contain the code necessary to perform the operation in the context of the application.

In applications that use the command pattern, typically *all* user-initiated application functions are made manifest by such objects. For example, in a drawing program, command object classes might exist for operations to draw figures, resize, delete, or otherwise modify figures, and to cut, copy, or paste figures.

An ordered sequence of instances of these objects is called a *command history*, because it represents each operation performed by the application during a session. Since each operation is captured, the history also represents the sum total of all of the states of the application during a session, up to and including the "present" state. Given a command history and an "uninitialized" application, the system can "replay" the operations in the history, and return the application to any state represented by the history.

To support generalizable undo and redo, the command pattern is extended so that operations can be run "forward" or "backward"—that is, they provide behavior that can invoke and reverse the operations they represent. Once this ability is added to the base command object pattern, command objects can be connected together in graphs to form complex histories that represent all of the possible states in which the application has existed. (See Thomas Berlage's excellent review of the command object pattern applied to history in [2].)

In most applications (including examples in the literature such as [12] and [16]), history is represented by a graph with a maximal vertex degree of one (more commonly known as a "line"), as in Figure 1.

Other systems [4] have a more complex model of time, in which history is represented as an arbitrary directed acyclic graph with a single root, as shown in Figure 2. Divergence in the timelines of such applications represent multiple plausible "alternate histories" that may coexist in the state of the application. Such applications must deal with issues of conflict detection and resolution, which do not occur in the simpler time models [5]. In both cases—linear and divergent histories—sequential sets of operations can be done or undone by traversing the graph.

A simple linear history of three nodes in a drawing program.

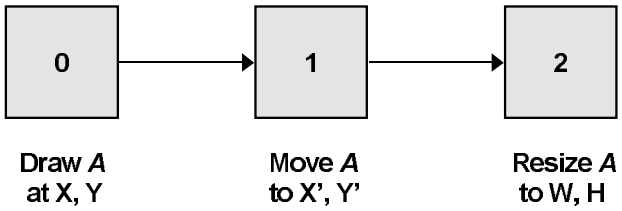


FIGURE 1: A Simple Timeline Represented as a Linear Graph

Timelines may exhibit multiple divergences and convergences.

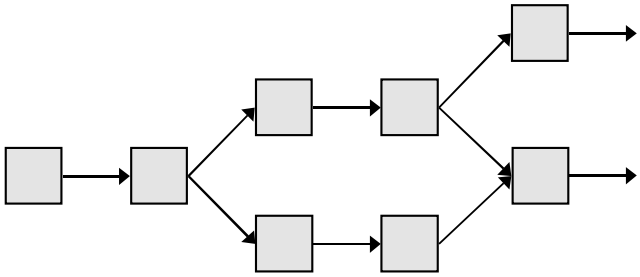


FIGURE 2: A Divergent Timeline

All of this material has been covered by previous research, but is necessary background for the next section. While prior work has explored timeline management in the context of traditional single-history applications, and even divergent multiple-history settings, it has not fully explored what might be called *interleaved* or *multi-level histories*.

The rest of this paper is organized as follows. The next section expands the rather simplistic "atomic" model of time presented here—in which each node in a history completely captures the results of some operation—with a more realistic model that accounts for operations with side effects by representing causal relationships in the timeline. This refinement of the simple history model has not been explored in the literature, and is a necessary extension for a wide class of "real" applications.

After this, the paper introduces the notion of multi-level timelines. These are timelines in which history is represented as a hierarchical decomposition of interrelated time streams. The work here unifies the more robust, causal representation of operations with this multi-level model of time.

Finally, the paper concludes with a discussion of several additional related topics—in particular, how applications can make such complex models of time efficiently searchable—and a set of conclusions and possible future work.

CAUSALITY AND SIDE EFFECTS

The simple history model described above is appropriate for applications in which the nodes in a timeline do, in fact, fully capture the state of the application, and can be invoked and reversed atomically. But is this always the case in "real world" applications?

Often, and as we shall see, the effects of a particular operation cannot be known a priori, and therefore the expression of such operations in the command idiom of time raises problems. This is best illustrated through an example.

A Case Study: Flatland

The impetus for much of the work described in this paper was a computer augmented whiteboard system, called Flatland [8][14]. Flatland presents a user model in which a whiteboard is loosely subdivided into regions of activity called *segments*; each segment represents some task on which the user is working.

In Flatland, segments can contain simple "raw" strokes that are unprocessed by the computer, or they can have application-specific *behaviors* added to them that can process the strokes. These behaviors can modify the interpretation and display of strokes in a domain-specific manner, and they can be flexibly added to and removed from a collection of strokes as desired. For example, a user may begin jotting notes in a "raw" segment, and then later decide to treat the information there as a to do list. By applying a "to do" behavior, the system will reinterpret the strokes as a structured list that allows the user to easily reorder and delete items. Figure 3 shows an image of a Flatland whiteboard containing a few segments.

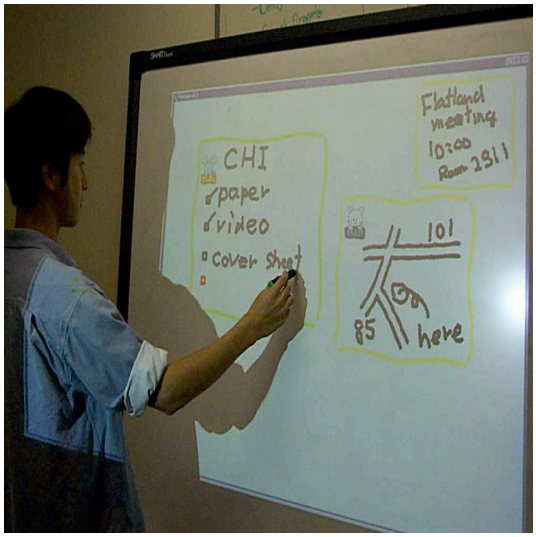


FIGURE 3: The Flatland Whiteboard Application

Behaviors have complete control over the state of the segment, and can even remove user strokes and add new strokes as needed. For example, a "map drawing" behavior lets the user draw strokes that correspond to streets. After the user draws a single stroke, the map behavior removes the original input, replacing it with two parallel strokes that represent the street. Intersections between streets are handled appropriately. Figure 4 shows an example of this map

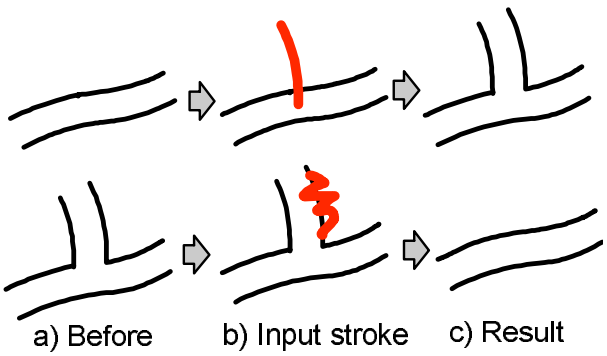


FIGURE 4: An Example of a Complex Behavior

For the purposes of this discussion, the key point about the Flatland architecture is that the system allows arbitrary, pluggable bits of application code to be dynamically bound to particular regions of the whiteboard. The *actual* operations that take place in a segment are not dependent just on user input, but also the behaviors associated with the segment, and their respective states.

The Problem of Side Effects

The Flatland design presented some problems that prevented its state from being accurately represented as a linear graph of command objects. In "traditional" uses of the command object idiom, each command is atomic—that is, it can reliably and completely do or undo its operation, and has no side effects that aren't represented by the state in the command object itself. As an example, when a command object in a drawing program is rolled forward, it must take care to store all information needed to completely reset the state of the application if it is rolled back.

In simple terms, no operation may make changes that would be impossible for it to undo, and thus each command object must be fully "aware" of the semantics and implications of the updates that it performs on the state of the application. To revisit the drawing program example, if performing the operation causes some change to be made to the graphics context of the application, the creator of the command object must be aware of this side effect, and must account for it when performing the corresponding undo. All of these possible side effects must be known *at the time the set of command classes are written*.

This situation is in contrast to the basic architecture of Flatland, and to many other applications in which side effects can not always be known or computed a priori (including, for example, Kramer's Translucent Patch system [11]). In Flatland, the use of extensible, pluggable behaviors means that essentially *every* interesting update to the state of the application *does* occur as a side effect to user input. The set of operations that can occur when a user draws a stroke on the board is dependent on the set of behaviors installed, and the current state of each of those behaviors.

Likewise, other applications—whether because of difficulty of implementation or core design issues—may not be able to fully know and express the consequences of each operation at the time the set of command objects is created. This leads

to problems in applying the command object idiom in the face of unknowable side effects.

Two Approaches

One approach to solving this problem would be to represent only the original user input in the command history. So, to use Flatland as an example, if a user made a stroke, and then the map behavior erased this stroke and replaced it with two parallel ones, only the original stroke (which doesn't even appear on the screen, after the map behavior is finished with it) would be present.

The benefit of this solution is that only "known" types of commands are stored in the history. The downside is that the history no longer represents the complete state of the application—in other words, the history is no longer a "self describing" representation of the application's state, since the command objects themselves do not fully capture the updates to the application. To jump to a different node in the history graph involves essentially "replaying" the user input to the individual behaviors, causing them to perform all of the same operations that they would in response to "fresh" user input. The computations done by behaviors can be arbitrarily complex, which means that jumping to distant points in the timeline can be arbitrarily expensive. (And the state changes of the behaviors themselves aren't represented in the timeline, so there is no way to search for them using the techniques presented later in this paper.)

An alternate approach would be to require that any behavior that updates the application's state express its updates in terms of new command objects. So in the example of the map behavior, the history would contain a draw of the original stroke (added by the user), followed by a removal of that same stroke, and draws of two additional strokes (added by the map behavior). This approach has a big advantage: changes based on user input are "pre-computed" by the behaviors, and only their final outputs are represented in the history. Essentially the "side effects" of the input are turned into "foreground effects" and represented as first-class citizens in the history. In this model, the history is once again a complete, self-describing representation of the states of the application.

Flatland follows this second approach. We believed that the benefits of a self-describing representation of history were apparent. First, by pre-computing as much state as possible we could allow efficient navigation along the timeline. Second, by representing all "user visible" output in the timeline, we could more efficiently support search over the timeline (see later in this paper for the details of search).

Of course, such a model subtly changes the atomicity constraints of the timeline. While individual nodes are still atomic, many of these nodes exist in the timeline *solely* because of causal effects and, in essence, cannot "stand on their own" independently of the operations that caused them. This means that a stepwise navigation of the timeline can move the application to states in which the necessary causal effects of some operation are not fully realized. The next section explains this in more depth, and presents a model for representing causality in the timeline.

A Transaction Model for State Changes

As mentioned before, in an application without side effects such as those that arise from the extensible behaviors in Flatland, each command object represents an atomic operation. You can redo or undo individual commands, and set the application to a known and sensible state. But if operations exist in the timeline solely because of the side effects of other operations, then this is not the case.

Consider once again the map behavior example from Flatland. Suppose that a user has drawn a stroke that corresponds to a new road, and then needs to roll time back. The original stroke command is actually turned into four separate command objects by the behavior—the original draw, a removal, and then two draws to render the road. Clearly, rolling back atomically is probably not what the user wants to see: a roll back would reveal the individual low-level operations of the map behavior, rather than the semantic "chunk" containing the whole set of operations associated with drawing a street.

For this reason, we developed a transaction model for the command objects in our histories. Sets of commands are grouped together into transactions that reflect causality, and transactions can be nested to an arbitrary depth to represent nested causality. In the classic database model, transactions provide a way to group related operations and provided the so-called ACID properties (atomicity, consistency, integrity, and durability) [9]. Flatland transactions are considerably lighter weight, and we do not want to imply that they maintain all of the rigorous consistency and durability guarantees of database-style transactions. Instead, Flatland transactions provide a mechanism for grouping operations into cause and effect relationships.

Each original user-level input begins a new *top-level* transaction. As the Flatland event dispatch machinery begins its invocation of any attached behaviors, a new nested transaction is started to "collect" the results of the behaviors. The behaviors perform their operations by adding new command objects to the history; these command objects are grouped into this new transaction. If they in turn call out to other behaviors, or other behaviors are implicitly invoked by the operations they perform, a new nesting scope is created.

From this model, causality relationships are clearly indicated: each nested transaction is executed as a result of the operation in the immediately higher scope. Transactions are represented explicitly in the history by `OpenTransaction` and `CloseTransaction` command pairs, and the history roll-forward and roll-back machinery is augmented to process transactions in whole, atomic, increments.

Figure 5 shows an example of the use of nested transactions to represent causality in the timeline. In this example, a roll-forward or roll-backward would "consume" the entire top-level transaction T_A, and all of the "side effect" operations represented by T_B and T_C. (Any other top-level operations that might exist at nesting depth zero—that is, stand-alone operations not a part of a transaction—would be consumed on a per-operation basis, as before).

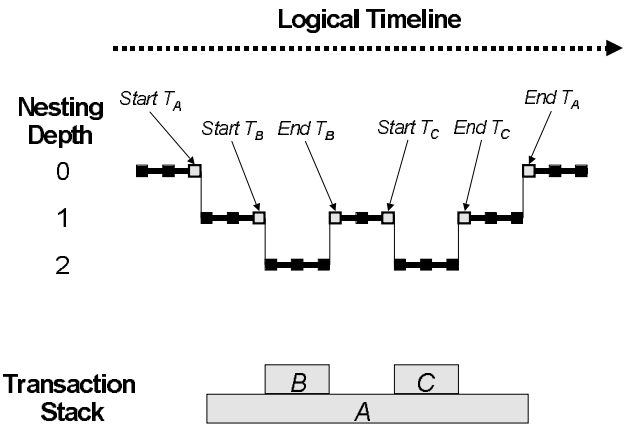


FIGURE 5: Nested Transactions Represent Causality in a Timeline

The model presented here has some similarities with hierarchical event systems [10], but there are a number of key differences. First, hierarchical events systems provide a *bottom-up aggregation* of multiple low-level events into more semantically meaningful high-level events (a sequence of mouse clicks and keypresses might become a "Save File" event, for example). In contrast, systems like Flatland are *top-down*: a single user input has rippling side effects that aren't—and can't—be known at the time the command object representing the input is created. Transactions in our case are used to preserve causality; hierarchical events serve to impose interpretation.

A second difference is that hierarchical events, in implementations such as Kosbie's and Myers', can fruitfully allow users to undo operations at a number of levels (undoing the last character typed in a Save File dialog, versus undoing the entire high-level Save File operation, for example). This ability to undo fine-grained operations within a hierarchy is appropriate for such systems, because at each granularity, nodes in the history still represent operations explicitly performed by the user. Contrast this to Flatland-like systems in which only the initial operation corresponds to an input made by the user, and the other operations are generated as a part of the system's implementation. We felt that, given our model, preservation of causality during undo was essential.

Consequences of Side Effects on Timelines

But what are the implications of such nested transactions on the basic timeline representations, and the operations available for operating on timelines? As it turns out, for all of the cases discussed so far—essentially simple, linear timelines—transactions do not alter the basic logic of timeline manipulations.

In all of these cases, a transaction still represents what is *logically* an atomic operation—even though it may comprise multiple constituent operations, these cannot be executed independently of their transaction-mates, and the operations in a transaction must still complete as a whole, if they complete at all. The transaction machinery described here is necessary to capture and correlate the causal relationships of side effects in the applications that may exhibit them. But the presence of these transactions does not fundamentally change the basic temporal model; essentially you can think of a set of operations within a transaction as reducing to a single logical operation, and the fundamental model holds.

Figure 6 shows a logical transformation from a simple, linear timeline which uses transactions to represent causality to an isomorphic one in which transactions are "collapsed" into a timeline basically identical to the one in Figure 1. As long as the roll-forward and roll-backward machinery is augmented to move through the timeline on transaction boundaries (which become essentially the "new" representation of atomicity in the system), then nothing else need change and the basic model holds.

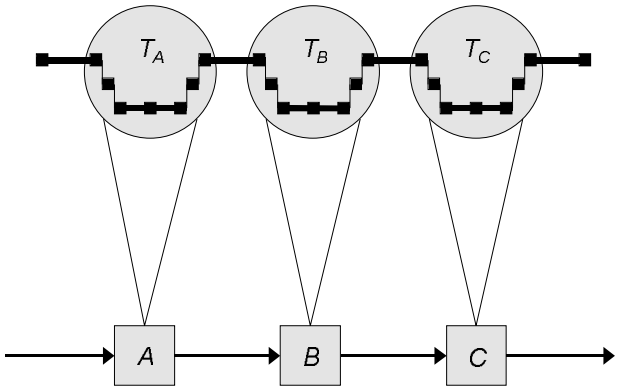


FIGURE 6: Timelines with Transactions Collapse to Simple Timelines in the Simple Case

As we shall see, however, this is not the case in more complex timelines, including the "multi-level" timelines described in the next section.

MULTI-LEVEL TIMELINES

The previous section presented an extension to the simple timeline model that can accommodate applications in which the side effects of operations may not be known when the command objects representing those operations are created. As you saw, while the use of transactions makes it possible to extend the timeline model to such applications, it does not fundamentally alter the logic of these simple timelines.

The presence of transactions *does*, however, have consequences in more complex models of time. The previous section addressed only the use of transactions in linear (non-divergent) timelines that have a *single level* of temporality. This section presents the notion of *multi-level* or *interleaved* linear timelines, and shows how the transaction model can be extended to accommodate it.

An Example of Multi-Level Time in Flatland

Once again, the notion of multi-level time was motivated by our experiences with Flatland, so we shall use that application as an example. One final timeline management issue we had to deal with in Flatland was the distinction between the "local" timelines of individual segments and the "global" timeline of the entire board. We wanted the ability for users to interact with the timelines of individual

segments, as we felt that this would be the most common style of history management: undoing and redoing changes in a single region of the board. By "local" timeline, we mean that we wanted to give the illusion that each segment had its own separate timeline. So if a user operated on segment A, went to a different segment, and then came back to working with segment A, undo operations in A should "skip" any operations done in other segments. For the purposes of local rollback, the entire history of A should appear logically continuous, even though in "real" time, operations on other segments may be interspersed with operations on A.

But we also needed the ability to roll forwards and backwards in global (whole board) time. Global undo and redo means that operations are undone and redone in their global, real time order, no matter which local segment they may be associated with. Such traversal of the global history is useful for reverting the entire board to an earlier state, say, to recover the context from a previous meeting.

In terms of the user interface, local undo and redo are performed by making particular marking menu strokes on individual segments; global undo and redo are performed by making these same strokes on the "root" segment (the whiteboard's background area).

Recall that segment histories can be arbitrarily interleaved from the global perspective: users can visit and leave segments as often as needed, causing the "real-time" history to jump from segment to segment. Because of this, the global history can be thought of as portions of the individual segment histories "packed" or "interleaved" together. In the Flatland implementation, each segment does, in fact, maintain its own contiguous, "segment logical" history; the global history timeline is an illusion created by stitching these individual segment histories together.[1] The particular representation used by the Flatland implementation is that the global history is stored as a list of "chunks." Each chunk represents one portion of the global history that occurs in one segment. The chunks contain a reference to a particular segment, as well as start and end indices for the range of commands within a segment history that fall into this particular place in the global history.

This example illustrates the use of a complex, multi-level model of time. The model presented to the user is a two-tier arrangement in which segments are contained within a larger surface, any of which can be manipulated independently. The histories at each level are separate from the perspective of user interaction, but yet still logically interrelated. Figure 7 shows such an example. Here you see three local timelines; a global timeline stitches these local timelines together, producing a representation of the "real-time" history of the entire artifact.

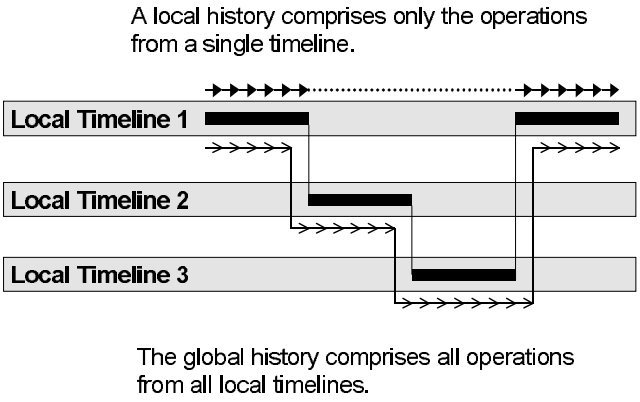


FIGURE 7: A Global Timeline Comprises Multiple Local Timelines

In general, such a model of multi-level time is appropriate for any application in which there are nested artifacts that—while they are related—users may wish to manipulate independently. In a source code control system, for example, users may wish to have fine-grained control over individual subprojects, and yet still be able to traverse the timeline of the entire source base, say, to look at an earlier release snapshot. In such an application, the individual histories of the subprojects are, of course, related to the larger history (and vice versa). But by allowing them to be manipulated independently, we allow users the freedom to contextualize the artifacts in the application appropriately. They can, for example, freely traverse the timeline of a particular subproject while keeping the rest of the global timeline in its most recent state.

Interleaving in the Simple Case

In applications that can support timeline models with simple atomicity—that is, without the need for transactions to indicate causality—the interleaving of timelines from different levels of a hierarchical history does not present significant problems. Since each node in any timeline is atomic, a plausible global history can jump from local timeline to local timeline without affecting consistency.

To put it another way, there is no point within a local timeline at which it is "illegal" to jump to another timeline. In the context of Figure 7 above, segment histories can be "mixed and matched" in any way to achieve a global history, since in a purely atomic system no operation may have side effects not completely expressed by the command object that represents the operation.

Interleaving and Transactions

Unfortunately, in a system *without* simple atomicity—that is, one that requires transactions to indicate causal effects—this is not always the case. Recall that when transactions are used to group related operations, transactions must complete

1. In the Flatland implementation, local segment histories are stored directly and the global history is maintained as a "shadow" data structure that references those histories. Other applications could take the opposite approach however, by maintaining the global history as the "real" representation and deriving local histories from it as needed. We took the former approach in Flatland because such an arrangement allowed us to "fault in" individual segments from persistent storage, without having to bring in the entire timeline. So, while Flatland creates the illusion of a global history timeline by composing individual segment histories together, this implementation feature is not dictated by the model.

atomically. Thus, a global history cannot be constructed by "jumping" from one local history to another at any point that is inside a transaction. Consistency can be maintained *only* if such jumps happen on even transaction boundaries.

In the context of the Flatland implementation, some operations naturally have effects in multiple segments. For example, split and join operations, which necessarily affect more than one segment at a time, produce operations in the timelines of two involved segments. In essence, an operation in one local timeline can produce operations in other local timelines and, hence, the global timeline as a whole.[1]

Problems arise when a transaction is in effect in the original timeline while modifications are made to other affected timelines. For example, suppose a Flatland user working in a segment makes a gesture that means that the system should "promote" any selected strokes in the segment to their own, new segment.[2] This is analogous to copying and pasting a set of strokes into their own window, created to hold them.

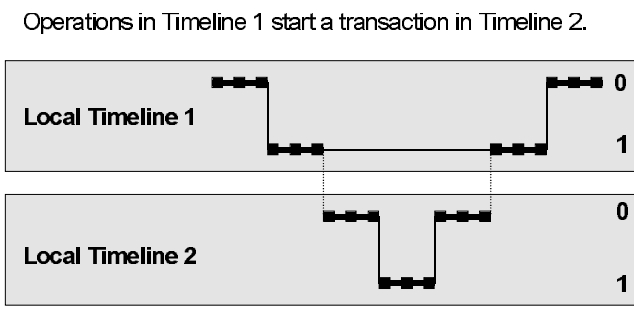
The initial operation begins a new toplevel transaction in the original segment. The "promote" operation is added to its timeline to indicate the start of the operation. After this a *new* segment is created as a result of the operation, and its timeline is initialized by a "drawStroke" operation that renders in it a set of initial strokes. The creation of the segment and drawing of the strokes occur as side effects of the execution of the promotion operation.

In this case, the timeline of the original segment is modified, and the promotion behavior causes the timelines of *other* segments to be modified as a result of its processing. In the "real-time" history—meaning the actual ordering of operations that occur on the whiteboard—a transition between the two segments occurs *during* an open transaction on the original segment. Figure 8 shows an example of this. Here, there is a top-level transaction in timeline T_2 that occurs during (realtime-wise) an open top-level transaction in timeline T_1. Clearly, there is a question here of how to piece the global history back together when the local histories don't cleanly separate on transaction boundaries.

From the local perspective, the operations in T_2 constitute a toplevel transaction, since there are no enclosing operations in that timeline that bracket them. But these same operations, seen from the global perspective, *are* enclosed in a transaction, since they were caused by the operations in the original timeline, T_1.

1. Actually, it is the combination of transactions with the use of operations that can have effects in other timelines that causes problems with interleaving. Applications that do not provide multi-timeline operations do not see the problems described in this section, even though they may use transactions to represent causality.

2. This is a hypothetical example based on Flatland. While Flatland does support a number of operations that have side effects in multiple segments, these operations (including splitting and joining segments) are more complex than the example given here and would require a more intimate knowledge of the Flatland architecture to explain.



Note that the transaction in Timeline 2 occurs while an open transaction still exists in Timeline 1.

(Numbers on the right indicate nesting depth.)

FIGURE 8: Operations Can Have Effects in Other Timelines

Before considering how to traverse the global timeline, though, it is important to first note a property of the local timelines. Even though the real timeline leaves one segment and visits another during an open transaction, transaction boundaries are still intact from the perspective of the individual segment histories. That is, the histories of both the original segment T_1 and the newly created segment T_2, viewed on their own, present an uninterrupted and contiguous sequence of transactions. These local timelines can be traversed, exactly as before.

The difficulty comes with reconstructing the global timeline. In all of the examples presented so far, the global timeline essentially *is* the real-time timeline. That is, the stitching together of individual, local histories produces a global history that is exactly the same as "what really happened" in real-time.

In this case, however, "what really happened" does not correspond to a logical, transactionally-correct global history. Two local histories contain top-level transactions that do not align with each other evenly. The global series of operations opens a transaction in one timeline and then visits another before the transaction is completed. Only after returning to the original timeline is the transaction closed and the entire operation finished.

The problem is that, in such cases, causality *spans* timelines. A cause in one timeline has effects in others, all of which must complete before the initial causal event can said to be completed. In other words, our simplistic model of transactions occurring—and completing—solely within single timelines, is over-simplistic. It cannot accurately represent side effects that span timelines.

Such timeline-spanning side effects, while difficult perhaps to envision, are actually fairly common. In a source code control system, an operation that moves a source file from one subproject to another has effects in the timelines of each. And, just like in the Flatland example, causality in this case spans across a timeline boundary.

A Possible Approach

One possible approach to this problem is to essentially situate transactions in the global timeline, rather than the individual local timelines. That is, transactions would be global entities, and could span across multiple local timelines. In essence, this would turn the case illustrated in Figure 8 on its head. The global timeline would contain a toplevel transaction, and the operations that constitute that transaction would be interspersed across both local timelines.

The problem is that such an arrangement destroys the semantics for local timeline navigation. The causal relationships in the global structure are repaired, but at the cost of sacrificing causality in local timelines—local transaction boundaries are completely lost, and the history traversal machinery cannot determine where to start or stop during a roll forward or roll back to reflect causal relationships. Well-formed transactions, in this model, do not exist in the local timelines, because the operations in a transaction skip across the various local histories.

Instead, what is needed is an arrangement that preserves causal relationships in local timelines by ensuring that local timelines consist only of sequences of complete, well-formed, top-level transactions (and the nested transactions contained within them). This must happen while ensuring that the global history can be constructed in a way that is "transactionally-correct"—meaning that it consists of ordered sequences of the top-level transactions from the individual local histories.

A Solution

The question here is really one of semantics. Either transactions span timelines or they don't, but in either case we need to ensure that both global and local traversal act as expected and preserve the necessary causal relationships—that is, by hiding side effects that show up merely as a result of the way the application is architected.

When traversing time in a local timeline, the existing transaction boundaries should be preserved. So, in the promotion example, traversing the timeline of the original segment T_1 should cause the undo or redo of all operations in that timeline contained within the promotion transaction. Operations in other timelines should *not* be invoked since, after all, the point of doing local undo and redo is to not affect other timelines. In essence, the traversal of the local timeline happens "outside" of the global history.

The second timeline in this example, T_2 (the one belonging to the newly-created segment) should likewise experience "normal" traversal of its local timeline on transaction boundaries. Even though the initial transaction in this segment was caused by an operation in another timeline, it is entirely conceivable that users may wish to roll back the operations in the new timeline, independently of the timeline of the original segment. Again, local traversal is defined to only "see" operations in the local timeline, and traversal happens on toplevel transaction boundaries.

For the global case, the operations in the two local timelines cannot be logically separated. That is, to roll back the global timeline necessarily requires that the undo of the promotion operation undo its effects in both timelines. This is, after all,

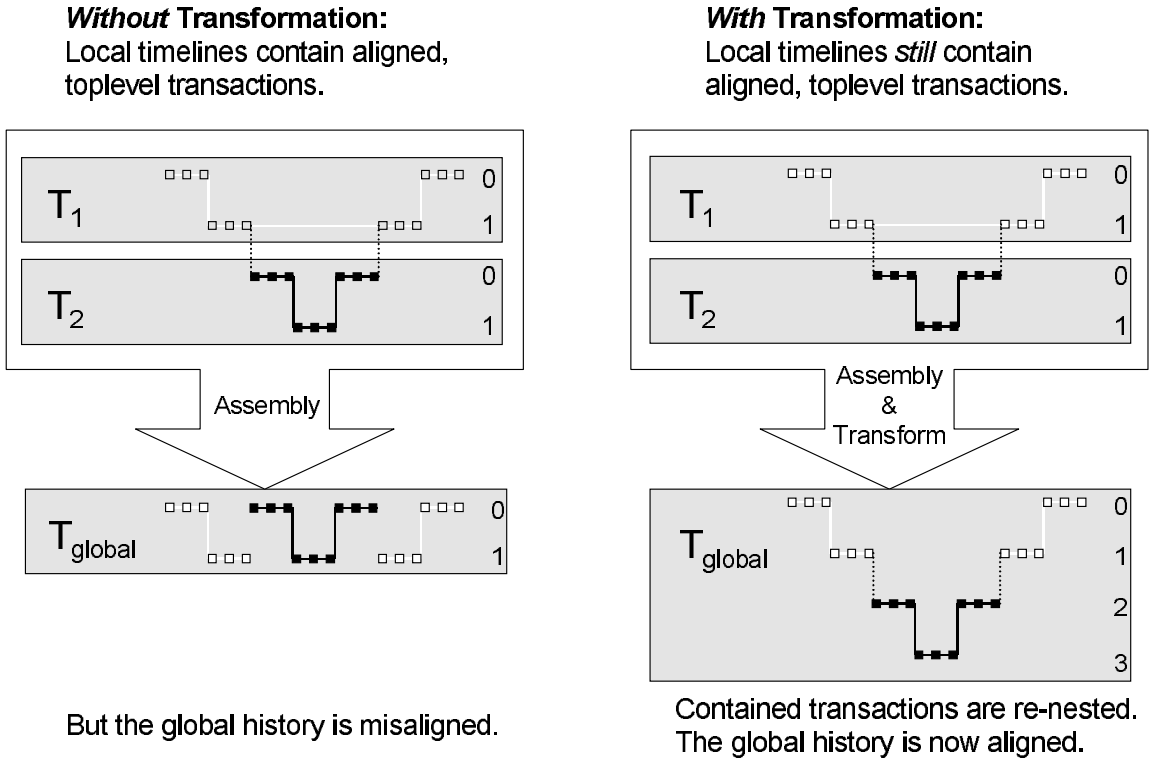


FIGURE 9: A "Push" Transformation Moves a Toplevel Local Transaction to a Different Global Nesting Level

the definition of global traversal—that the effects of a roll forward or roll back span the local timelines, and therefore should effect changes across all local timelines.

The desired behavior can be achieved by a transformation on the global timeline, to "repair" the damage caused by non-aligned transactions. First, the system can identify toplevel transactions from a local history that are globally nested within a transaction from another local timeline (such as the transaction in T_2 in Figure 8). In the global timeline, these transactions are then "pushed" to a higher nesting order. Essentially, what is a toplevel transaction in a local timeline becomes a nested transaction globally.

Figure 9 illustrates this transformation. Here you see what happens both in the absence of the push transformation, and with the push transformation. On the left, without transformation, the two local histories are assembled into a misaligned global history—two toplevel transactions from different histories are interleaved. On the right, the transaction from T_2, even though it is "toplevel" from the perspective of its local history, is causally related to the transaction in T_1. So it is pushed to a greater nesting depth when the global history is assembled.

This gives us the desired behavior in cases like the one illustrated in this section, where causal effects across timelines cause transaction boundaries to misalign globally. Local timelines still consist of a continuous stream of toplevel transactions, as they did before. This allows roll forward and roll back to occur in a local timeline while preserving all causal relationships that originate in that timeline. And yet the transformation establishes causality *across* timelines, necessary for global traversal; global roll forward and roll back may undo or redo operations that logically span local timelines.

In the Flatland implementation, there is no change in representation to accommodate these new semantics. Instead, an *ex post facto* transformation is executed during reconstruction of the global history that identifies and "pushes" toplevel transactions from a local history that are globally nested within a transaction from another local timeline to a higher nesting order. This change isn't represented directly in the local timelines (in our implementation), but is produced on the fly as needed when the global history "chunked" data structure is produced.

Local versus Global Timeline Management

As mentioned before, multi-level timelines allow users to alter the state of a portion or region of some artifact, independently of the global timeline of the artifact. In essence, it allows users to make local changes in a "bubble" that is isolated from the larger global timeline. It does this by providing separate, but related, timelines for the artifact as a whole (the global history), and for the various parts of the artifact (represented as local histories). This ability is supported because local histories can be made logically and causally independent of each other, and of the global history as a whole.

We have not, however, yet investigated the inverse mechanism—holding one local history fixed while manipulating the rest of the global history. Such an operation may be useful as a tool for recontextualization. A user could keep one bit of an artifact fixed in time, and "scroll" the rest of the artifact forward and back to locate desired states. This would allow a user to focus on a particular subcomponent of a larger artifact, say for editing, while freely browsing earlier states of the remainder of the artifact.

In the Flatland design, in which these ideas were implemented, a change to the global timeline "snaps" all local timelines to the current global state. That is, while a local timeline can be freely manipulated independently of the global timeline, manipulations of the global timeline carries all local timelines with it.

Fully investigating isolation in both local and global timelines is an area for future research.

SEARCH OVER TIMELINES

At several points, this paper has mentioned the utility of being able to not only traverse the history of an application, but also to search it. Search is the ability to quickly scan the timeline to produce likely matches for some query. A number of user interfaces could be imagined for such a facility, ranging from textual query to visual matching.

The Flatland implementation uses two primary mechanisms for search. The first is efficient visual scan. This means that users can very quickly (in real time) "scroll" the application's state back and forth. The visual scrolling is supported by animation, and by "semantic snapping" in the time domain. That is, certain points in the timeline deemed to be "interesting" are flagged, and are somewhat easier to reach. Points when the user switches working from one segment to another are so flagged, for example.

Whereas the first search mechanism is more a tool for on-line browsing than off-line search, Flatland provides a second mechanism that "lets the system do the work." Users can graphically specify attributes of desired segments—it contained a map, it was mostly blue, it was on the left hand side of the board, and so on—and the system will produce a set of thumbnail snapshots of the segments that matched the search criteria at any point in their histories.

These are only two examples; clearly, other application domains will have other mechanisms for search. But the key to effectively supporting such operations is that the command objects in the timelines must contain very efficient representations of application state. As mentioned in the section on causality, one approach to providing a history system would be to "replay" the command objects to application code that then reinterprets the operations. But since application code may make arbitrarily complex (and expensive) responses to such commands, such an approach does not lend fast, deterministic searching.

Instead, in Flatland, command objects contain largely "primitive" operations which can be executed quickly to create a visual representation of the state of a segment. The search thumbnails are created by quickly scanning the history to only draw out those operations that affect the graphical presentation—and hence, are needed to create the

thumbnail. If a user selects a thumbnail, then the full history traversal machinery can be brought to bear to completely move the application to a new state.

One approach—which we did not take but would be a requirement in more scalable systems, or systems with longer histories—would be to periodically create a "checkpoint" of application state at various points in the history. Then, to move between distant points in the history would not require evaluating every command object in between the source and the destination, which is an *O(n)* operation for histories of *n* nodes. Clearly, we'd like better than linear performance for traversal and search.

To traverse to a different state in a checkpointed application, the system would load the checkpoint state from the node nearest the destination, and then do "normal" traversal the rest of the way to the destination. If checkpoints are produced regularly, say, every 10 nodes in the history, then any traversal or search is a constant time operation.

Again, we have not explored such an implementation in our work, but feel that the architecture is straightforward, and does not break or the basic timeline model presented here.

CONCLUSIONS AND SUMMARY

This paper has presented two extensions to the command object paradigm for representing and manipulating application histories. First, we have explored a mechanism to more fully capture causality in histories. Such a mechanism is essential for applications that cannot fully know the side effects of all possible operations *a priori*. Flatland and Translucent Patches are exemplars of this style of application, although any number of systems can use the mechanism presented here to relieve the burden of having to produce fully atomic and isolated sets of command classes.

Second, this paper has explored the notion of multi-level timelines. Such timelines are useful when an application presents an artifact that can be decomposed into constituent pieces. Multi-level timelines allow users to interact with the history of the complete artifact, or with the histories of the individual pieces. Essentially, it provides a model for being able to traverse the timelines of subregions of an artifact independently of the artifact as a whole.

More fundamentally, this work begins to flesh out the space of timeline manipulation systems described in the literature. Current systems can be described along three dimensions:

- Linear histories versus divergent histories.
- Simple atomicity versus causal atomicity.
- Single-level versus multi-level timelines.

Prior art has focused primarily on the first dimension, linear versus divergent histories. This previous work has assumed the existence of simple atomicity and single-level timelines.

The work here expands on the prior art to include causal atomicity in either linear or divergent histories. It also introduces the notion of single- versus multi-level timelines, and presents a model for multi-level timelines in the presence of either simple or causal atomicity. This work, however, only focuses on multi-level timelines for linear histories. The use of non-linear multi-level timelines has yet to be explored, in either the simple or causal settings, and is a topic for future research.

REFERENCES

[1] Berlage, T., and Genau, A. "A Framework for Shared Applications with a Replicated Architecture," *Proc. ACM Symposium on User Interface Software and Technology*. Nov. 1993.

[2] Berlage, T. "A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects," *ACM Transactions on Computer-Human Interaction*, 1:3, Sept. 1994.

[3] Dix, A., Mancini, R., and Levialdi, S. "Communication, Action, and History," *Proc. ACM Conference on Computer-Human Interaction*. March 1997.

[4] Edwards, W.K., and Mynatt, E. "Timewarp: Techniques for Autonomous Collaboration," *Proc. ACM Conference on Computer-Human Interaction*. March, 1997.

[5] Edwards, W.K., "Flexible Conflict Detection and Management in Collaborative Applications," *Proc. ACM Symposium on User Interface Software and Technology*. Oct. 1997.

[6] Freeman, E., and Fertig, S. "Lifestreams: Organizing your Electronic Life," *AAAI Fall Symposium: AI Applications in Knowledge and Retrieval*. Cambridge, MA. November, 1995.

[7] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts. Addison-Wesley, 1995.

[8] Igarashi, T. Edwards, W.K., LaMarca A., and Mynatt E. "An Architecture for Pen-Based Interaction on Electronic Whiteboards," *Proc. International Working Conference on Advanced Visual Interfaces*. Palermo, Italy. May, 2000.

[9] Korth, H. and Silberschatz, A. *Database System Concepts*. McGraw-Hill, 1991.

[10] Kosbie, D., and Myers, B. "A System-Wide Macro Facility Based on Aggregate Events," in *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.

[11] Kramer, A. "Dynamic Interpretations in Translucent Patches," *Proc. International Working Conference on Advanced Visual Interfaces*, Gubbio, Italy, 1996.

[12] Kurlander, D. and Feiner, S. "A History-Based Macro by Example System," *Proc. Symposium on User Interface Software and Technology*. Nov. 1992.

[13] Myers, B. and Kosbie, D. "Reusable Hierarchical Command Objects," *Proc. ACM Conference on Computer-Human Interaction*. April, 1996.

[14] Mynatt, E., Igarashi, T., Edwards, W.K., and LaMarca, A. "Flatland: New Dimensions in Office Whiteboards," *Proc. ACM Conference on Computer-Human Interaction*. May 1999.

[15] Rekimoto, J. "Time-Machine Computing: A Time-Centric Approach for the Information Environment," *Proc. ACM Symposium on User Interface Software and Technology*, Nov. 1999.

[16] Rhyne, J. and Wolf, C. "Tools for Supporting the Collaborative Process," *Proc. ACM Symposium on User Interface Software and Technology*. Nov. 1992.