CAUSALITY – A Conceptual Model of Interaction History

Mathieu Nancel
University of Canterbury
Christchurch 8140, New Zealand
mathieu@nancel.net

Andy Cockburn
University of Canterbury
Christchurch 8140, New Zealand
andy@cosc.canterbury.ac.nz

ABSTRACT

Simple history systems such as Undo and Redo permit retrieval of earlier or later interaction states, but advanced systems allow powerful capabilities to reuse or reapply combinations of commands, states, or data across interaction contexts. Whether simple or powerful, designing interaction history mechanisms is challenging. We begin by reviewing existing history systems and models, observing a lack of tools to assist designers and researchers in specifying, contemplating, combining, and communicating the behaviour of history systems. To resolve this problem, we present CAUSALITY, a conceptual model of interaction history that clarifies the possibilities for temporal interactions. The model includes components for the work *artifact* (such as the text and formatting of a Word document), the system *context* (such as the settings and parameters of the user interface), the *linear timeline* (the commands executed in real time), and the *branching chronology* (a structure of executed commands and their impact on the artifact and/or context, which may be navigable by the user). We then describe and exemplify how this model can be used to encapsulate existing user interfaces and reveal limitations in their behaviour, and we also show in a conceptual evaluation how the model stimulates the design of new and innovative opportunities for interacting in time.

ACM Classification Keywords

H.5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

Author Keywords

History Systems; Undo; Conceptual Model; Paradoxes

INTRODUCTION

Supporting the ability for users to Undo and Redo commands has been an expected part of interactive computing systems since the first graphical user interfaces [26]. The simple metaphor of allowing users to step back and forward through time enables users to revert to previous states and encourages exploratory learning because erroneous actions can be easily reversed or redone. Other forms of history systems can make interaction more efficient by providing rapid access to recently used tools or data (e.g., [11]). Yet more advanced history systems allow sophisticated manipulations of temporal aspects of interaction, such as parallel editing of alternative versions of a document [27], recycling past sequences of commands or command parameters [12, 15], undoing commands within a portion of the document [8, 9, 25], and concurrent collaborative work [23]. In short, previous research on interactive history systems is extremely rich.

A number of these works address technical aspects of interacting with command histories, including how to visualize them [8, 12, 14] and navigate through them [22, 24]. However items of prior work generally address separated problems or functionalities. Combining these systems or their features in a real-world application remains challenging as they often rely on different and sometimes incompatible paradigms and models, or generate different effects for the same operation. Many also exhibit conceptual problems caused by their underlying model. As a simple example, most interactive systems 'forget' undone commands or previous document states when the user triggers an editing command after having performed undo. Consequently, pressing 'undo' then accidentally triggering an action will, in many systems, eliminate the possibility of returning to the pre-undo state. Therefore, although built primarily for error recovery, such systems can be surprisingly error intolerant. Yet this need not be the case – an interface may maintain the undone command, or the data state, or both. Through this paper we demonstrate a variety of counter-intuitive, illogical or restrictive effects that can arise with history systems, and we attribute them to the lack of an underlying model with sufficient expressiveness to adequately contemplate temporal interactions.

The key problem that we address is that designers and researchers currently lack the conceptual tools for supporting, combining and generating temporal behaviours. To solve this problem, we introduce CAUSALITY, a conceptual model of interaction history in which commands, application parameters and document states are described as a *causal* and *unforgetting* system that is both linear and branching.

CAUSALITY, as illustrated in Fig. 1, consists of five main components that model different elements necessary for complete coverage of temporal interactions. 1) *Artifact* stores the state of the work object at discrete points, and is represented by the green bar in Fig. 1. The middle images of the figure show how the artifact develops through time, although the images are not part of the model. 2) *Context* stores the application state at discrete points, including parameter settings and the state of user interface controls. It is represented by the grey bar. For example, in a drawing application, *context* might store the current colour and line-width, etc. 3) *Com-*

CHI 2014, April 26 - May 01 2014, Toronto, ON, Canada

ACM 978-1-4503-2473-1/14/04...$15.00.
http://dx.doi.org/10.1145/2556288.2556990

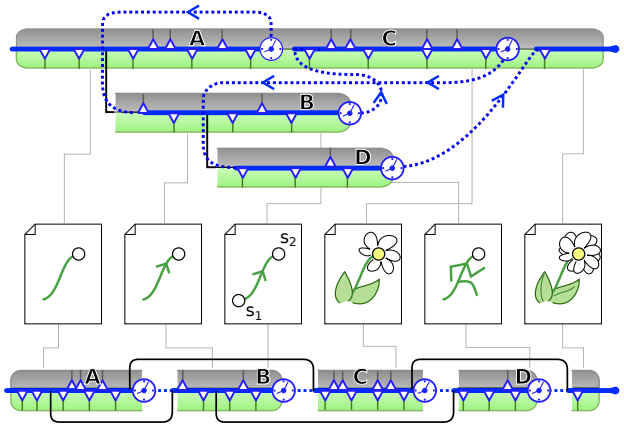

Figure 1. CAUSALITY overview: branching chronology (top 3 rows); linear timeline (bottom, and blue line in top 3 rows); lettered and illustrated states match in both visualizations. Application context is the grey bar; Artifact state is the green bar. Commands shown as blue triangles; time-travel as clocks and dotted lines within the linear timeline. Black strokes between states represent branching in both visualizations.

mands (blue triangles in the figure) cause state changes in the artifact (downward pointing) or context (upwards). For example, a drawing action constitutes a command affecting the artifact. In CAUSALITY, each command references all the elements of the artifact and context that it uses and affects. 4) *Linear timeline* stores the commands as a linear sequence of events (bottom of Fig. 1 and blue line in top 3 rows). 5) *Branching chronology* stores a tree-like virtual chronology of artifact and context, with each branch constructed by first using the interface to traverse through time (represented by the clock icons on the linear timeline) and then using a command to modify the artifact, context or chronology at that point.

The model's separation of *artifact* from *context* is particularly important. This is because some commands affect only the artifact (e.g., drawing), some only the context (e.g., changing line colour), and some both (e.g., cutting a drawn object). In general, the model (which includes other components described later in the paper) can encapsulate and describe existing interactive behaviours, and it also facilitates identifying and describing new possibilities for history interactions.

After reviewing previous research on temporal interaction from the HCI and general computing literature, we detail and exemplify the technical and conceptual issues that arise when interacting with time. We then fully describe the CAUSALITY model and explain how it solves these issues and how it can be used to implement the techniques and systems in the literature. Finally, we use a painting application scenario as a conceptual evaluation to show how CAUSALITY inspires powerful and exciting new interface behaviors. The contributions of the paper are (i) a comprehensive review of history in computing systems, (ii) an analysis of the limitations of prior models for history systems, and (iii) the provision of an exact set of 'thinking tools', instantiated in CAUSALITY, for designers and researchers to explore powerful and novel interactions with history. This paper presents the conceptual foundations of a larger project; specific implementations of CAUSALITY and their evaluation are left to future work.

HISTORY IN COMPUTING SYSTEMS

The following subsections briefly review key items from the extensive history systems literature, categorised by the main objective of each study: error recovery, command reuse, version management, and history models. In addition, many papers have examined interactive visualisations of past events for a variety of application domains (see [13] for a review). Although visualisations might be used to help the user navigate branching chronologies generated by CAUSALITY (described later), this review focuses on conceptual foundations, rather than their visual representation.

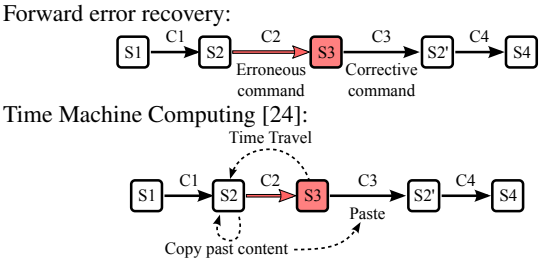

Figure 2. History of an artifact, with its different states (S) and commands (C). Forward error recovery (top) consists in applying corrective commands to recover from an erroneous action. TMC [24] helps this process by providing access to past versions of the artifact (bottom).

Error recovery

The primary goal of many history systems is to allow the user to recover from erroneous or mistaken actions. Abowd and Dix [1] distinguish backward and forward error recovery. *Forward* error recovery consists of performing additional editing actions to either cancel or correct the erroneous action(s) (Fig. 2). For example, using an eraser tool to cancel the effects of a drawing action, rather than undoing its creation. Rekimoto's Time Machine Computing [24] supports a variant of forward error recovery that allows users to reuse past data in modifying the current state (Fig. 2-bottom).

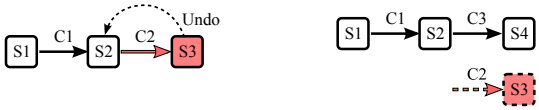

Figure 3. Undo (left) temporarily restores previous states of the artifact; undone commands are discarded if the user resumes editing (right).

Backward error recovery alters the command history of the artifact to restore past versions or cancel unwanted operations. The most common backward error recovery facility is linear undo/redo, often implemented with the 'Command' design pattern [10]. 'Undo' takes the work artifact back to its state before the last command (Fig. 3-left) and can be repeated to visit earlier artifact states. 'Redo' moves through time in the opposite direction, reapplying the commands, until the latter-most artifact state is reached. While undo is normally available in desktop applications for discrete commands influencing the data state, it has not generally been applied to mobile interactions, continuous interactions, nor for commands that affect view state (e.g., zooming). Recent research, however, has addressed these limitations: Dwell-and-Spring supports undo for direct manipulation; [2] and Loregian [16] examines undo on mobile devices.

The linear model of undo/redo systems has also been augmented in a number of ways to improve error recovery. *Selective Undo* [23] allows the user to cancel distinct commands without deleting all later history. The user is thus able to interfere with the history of the document instead of simply restoring previous versions chronologically. Chimera [14] and '3Rs' [4] allow the user to go back in time within a document or the whole system, applying changes to past data and/or commands, and then replay every later actions on the newly defined past. Chen *et al.* propose a subset of these functionalities in [8]: users can select past editing commands that required dialog windows and alter the parameters of these commands, then replay every later command.

However, modifying only parts of the history of a document can generate temporal paradoxes [19], which are conflicts in the course of actions that chronologically follow the undone or modified command(s). For example, if changing a past command results in a data object no longer existing, then every later command applied to that object is meaningless. Advanced history manipulations therefore require recovery mechanisms to detect and cope with paradoxes. Systems like Joyce [22] and Script [3] reject commands that generate paradoxes. *Cascading undo* [6] takes a different approach, tracking forward through every conflicting command, and triggering undo mechanisms to remove the conflict(s).

Command reuse

Accessing and reusing past commands can improve user performance, e.g. by repeating a command or by generating macros that apply complex operations in a single step. Kurlander and Feiner [15] described a system for selecting past sequences of commands and re-applying them to the current version of the document with different parameters. Their system identifies the required parameters of the macro, requesting input from the user; and it permits macro debugging. Bueno *et al.* [5] proposed a tool for collaborative editing of vector graphics, allowing users to merge work across different timelines. Chronicle [12] allows users to invoke dialogs from past commands with the same parameters.

Version management

Some history systems maintain alternative versions of a document, allowing users to explore different designs at the same time without needing to undo their previous edits. Subjunctive interfaces [18] allow the user to view and control alternative scenarios for a variety of uses including simulations and document editing. Terry *et al.*'s system [27] supports alternative document designs and provides interface mechanisms for quick editing and visual comparison. Chen *et al.* apply revision control mechanisms to binary files [8], enabling branching and merging similar to SVN.

History structure

Depending on their purpose, history systems record commands and states as either a list, tree or graph. *List* structures are the most widespread, recording artifact states and/or commands in a linear fashion, often chronologically [5, 21, 28]. This structure enables a variety of single-history operations, from Linear [10] to Selective [23] and Regional Undo [25]. *Tree* structures allow histories to diverge from a common past state, usually at the user's explicit command, so that several versions of a document can coexist in parallel. They enable exploration of alternative modifications of a single document [27] or the whole file system [22]. Finally, directed *graph* structures enable merging of parallel branches of history, such as those produced by multiple users [19] or from a single user's alternative designs [8].

ISSUES WITH HISTORY SYSTEMS

Although there is a wealth of prior research on history systems, most of this research has focused on meeting the requirements of a specific target domain. A smaller proportion of prior work has carefully specified the temporal model that underlies the behaviour of the systems disclosed, but again, when described, these models have generally been directed to the requirements of their intended domain or of specific functionalities. This specificity of application creates problems for designers who wish to apply features in new domains.

In this section we describe three main limitations of existing models of history: 1) they lack structural and functional *inter-compatibility*, which makes it challenging to combine their functionalities in actual applications; 2) they can cause *conceptually inadequate* effects, i.e. command results that go against the goals of the very system they support; and 3) their structure may *limit the vocabulary* of the possible history operations, and thus fail to support the user's intentions.

Inter-compatibility between history systems

As described above, the literature on history systems features a variety of functionalities including advanced error recovery, command reuse, history visualisation, version management, and underlying models. Although the total set of interactive behaviours described are tremendously diverse, there are substantial limitations in facilitating the reuse and combination of the methods disclosed. In particular, there are no tools to assist determining whether the techniques described in article A comply with the assumptions or methods proposed in article B. Example problems are described in this section.

Structural incompatibility occurs when two models cannot be merged. This can occur when the model's structure is incompatible (e.g., linear- vs. graph-based structures) or when the methods for linking commands within the structure are incompatible. An example of incompatible linking methods is given by the combination of Cass *et al.*'s [6] model, which links user commands along predefined process dependencies, with Parallel Paths [27], which logs alternative courses of events. While undoubtedly useful, neither of these systems maintain strict chronological order of commands, meaning that past commands (including operations on history) cannot be accessed or undone chronologically: they cannot trivially be combined with classic undo/redo. Critically, though, the ways that they deviate from chronological behaviour are fundamentally different, and therefore there are also complexities in their combination (resolved by CAUSALITY).

Functional incompatibility occurs when two systems react differently to seemingly identical commands. For example, when undoing a past command C: Linear Undo undoes all

commands more recent than C; Script [3] and 3Rs [4], in contrast, attempt to re-run the command sequence that follows C and will cancel the undo if it generates paradoxes; Cascading Undo [6] also undoes the commands that share dependencies with C. These effects are all reasonable consequences of an Undo command, so there are design complexities in modelling and combining these behviors in a single system.

Conceptual inadequacies

Conceptual inadequacies are system behaviours that oppose its own primary goals. We identify two main conceptual inadequacies in existing history systems, and they stem from limitations of the underlying data model: their unforgiving nature [22]; and illogical undo.

Unforgiving Undo

Many history systems were designed primarily for error recovery, yet they can be surprisingly error-intolerant. With linear undo histories, undone commands and the resulting history states are typically deleted when new commands are triggered (Fig. 3). The replacing commands can be undone afterwards, but the discarded piece of history cannot be retrieved, and therefore mistaken manipulations during use of these systems can lead to non-recoverable loss of work. For example, if the user realizes that she shouldn't have undone some commands or if she inadvertently triggered an editing command in the middle of an undo navigation. This problem also occurs with advanced history systems, like Cascading Undo [6]. The inadequacy is a consequence of the traditional linear modeling of history[1]: old commands and document versions must be deleted to make space for new ones.

By modeling history as a tree or a graph, some systems [19, 22, 27] can prevent parts of history from being mistakingly forgotten by saving them in a parallel subtree. However, the creation of a branch has to be triggered by the user and thus can only partially help with error recovery: undos can still delete commands or states that were not explicitly saved.

Illogical Undo

History models also cause illogical behaviors when dealing with non-linear undo of past commands. For example, the seminal Amulet system [20] enables Selective Undo by storing the previous value of a modified property in the Command object itself (the `OLD_VALUE` slot) and uses it in the `undo()` method of this Command object. As illustrated by the authors, "[if] a command turns an object from blue to yellow, selectively undoing the command will make it blue, no matter what its current color is" [20].

This is an example where a data model only supports the simplest cases of a technique's principle. For example, as illustrated in Fig. 4, say that a vector object has undergone three `setColor` commands A (from black to red), B (red → green) and C (green → blue); say also that the user selectively undoes B then C (not shown in Fig. 4). The last two color changes have been undone: one could expect that the object turns back to the result of A, i.e. red. However, following Amulet, the last undo command (on C) triggered `setColor(OLD_VALUE)`, causing the object to turn green, i.e. the effect of B, which was undone first.

[1]Cass *et al.*'s system [6] models *processes* as a tree, but modifications of this tree are applied linearly.

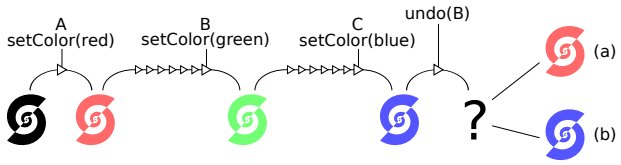

Figure 4. Possible effects of a Selective Undo: either restore a previous parameter value (a) or remove a command from history (b).

Limited vocabulary

History systems are usually designed to deliver the 'best' response to specific problems or functionalities. Consequently, their models often allow only one outcome to a user command even when other reasonable responses could be considered. Indeed, studies showed that the users' expectations to non-trivial modifications on history are not always fully consensual [7] and can depend on the platform [16, 17]. We illustrate this with two examples.

Case 1: Possible outcomes of Selective Undo

Using the previous `setColor()` example, say that the user selectively undoes only B, not C. Following Amulet, the system calls B's `undo()` method that is `object.setColor(OLD_VALUE)`, so the object turns red despite the later command C (Fig. 4-a). This outcome makes sense if the user's intention was to restore the object's color to before B was applied. However B could also have been selected and undone as part of a batch of commands corresponding to a higher-level editing process, in which case she would probably prefer this object to remain blue as the later (and not undone) command C set it (Fig. 4-b).

Case 2: Paradox resolution with Cascading Undo

In [6], Cass and Fernandes describe a scenario where selectively undoing the creation of a virtual section in a structured document results in automatically undoing the (later) generation of a table of contents, because their model logged a functional dependency between the table of contents and the section. Again this could be the user's intent, but she could also completely change the structure of her document yet still want a table of contents at its beginning. In this case, the proper outcome would be to update the table of contents to the 'new' structure. Both outcomes are valid depending on whether the user considers the generation of the table of contents to be a consequence of the creation of the section or not, and possibly more outcomes could be imagined, but the history model only enables one of them.

THE CAUSALITY MODEL

CAUSALITY is a new model of interaction history that reflects the causal system of a process along with its virtual chronology and real-time ordering. It provides persistent, transitive relationships between the user actions, their parameters and their consequences on the edited artifact throughout its history. We designed it in order to solve the problems described in the previous section while enabling most existing history

techniques and systems of the literature. Its structure also enables whole new paradigms of interaction histories, that we will illustrate in the next section.

The upcoming description of CAUSALITY is intended to be exact and precise, as is necessary for designers and researchers to be able to make use of it. Core concepts are described earlier, with details deferred to later subsections. A reader initially uninterested in the details required for replication might consider skim-reading once past the core concepts. However, we hope that such as reader will return to the details once inspired by the example applications of CAUSALITY presented in the following section.

Artifact and Context

CAUSALITY considers both the *artifact* (e.g. the edited document and its composing elements) and the editing *context* (i.e. the different states and values of the application's or system's parameters). The context is composed of elements that are used in editing operations but do not affect the artifact directly. For example, changing the foreground and background colors in Adobe Photoshop has no effect on the image but can affect the result of later actions such as painting. A command that influences the context may also concurrently influence the artifact – for example, when a text region is selected, clicking the bold button affects both the text in the artifact and the parameters in the context (toggling the bold state). In most cases the artifact will be a single-file document, and the context will relate to its editing software, but these notions can be extended: a Java project can be considered an artifact composed of its packages, files and binaries. Further, the entire file system can be considered an artifact [2, 4, 24], in which case the context parameters could be those of the operating system or of the file browser.

History is thus separated into the artifact history and the context history, which are composed of the various versions of the artifact and context linked by the user commands. The artifact and context histories describe the evolution of different datasets but are temporally connected: changing a parameter of the application generates a new state in the context history but not in the artifact history, and vice-versa. Some commands generate new states in both histories; for example, the 'Cut' command pushes new content to the clipboard (context) and deletes part of the file content (artifact).

Linear Time and Branching Chronology

CAUSALITY models history as a tree and uses *systematic branching*. When a command is applied to a history state S a new 'child' of S is generated, regardless of the number of children that S already has (Fig. 5-left). Similarly, every change to an existing portion of history (e.g. modifying, deleting or inserting a command C) generates a new subtree that replicates the original history but includes the changes; this new subtree thus starts with C and branches from the last state of history before C (Fig. 5-right). This way, the original sequence of commands is preserved. This duplication can happen any number of times from any state.

One consequence of using a tree-like branching chronology is that states that belong to parallel branches cannot be formally ordered, e.g. context states A and B in Fig. 1. However commands themselves can be ordered since they take place in real time. CAUSALITY thus defines a *linear timeline* that is the unalterable set of commands issued by the user as they occurred in real time. Linear time disconnects from the branching chronology when the user 'travels' in time, as illustrated in Fig. 5. The linear timeline thus contains every command performed by the user, including operations that modify history that we describe later in the paper.

Figure 5. Left: *applying* a command C_a to a past state always generates a new state of history regardless of its existing 'children'. Right: *inserting* a command C_i in an existing sequence of commands generates a new subtree, with replicated commands from the original branch (blue halo). The user travels to the end of the branch to see the effect of the insertion.

This allows commands or states to be retrieved based on either *real time*, e.g. "what I did before lunch", or virtual *chronology*, e.g. "before this paragraph was edited". Chronology-based navigation follows the order of states in the tree structure, regardless of when these commands were invoked in real time; time-based navigation, in contrast, follows the order of commands as they were invoked by the user regardless of the branching chronology tree structure. For example, in Fig. 1, the lettered context states happened in alphabetical order in real time; but in the branching chronology, it can only be said that A preceded C and that B preceded D. Any visualizations based on the linear time, e.g. Fig. 1-bottom, can be used to easily spot clusters of commands that correspond to specific editing processes, as with the Chronicle timeline [12]. In contrast, any visualization based on the branching chronology enables a rich view of actions upon different versions of the interaction context or artifact.

References as Versions or Subscriptions

The artifact and context are composed of elements that are modeled as encapsulated data structures composed of other elements and/or data fields, similar to member variables in Object-Oriented programming. Fig. 6 illustrates how a particular vector drawing artifact state might be composed of several elements, each of which having its own history.

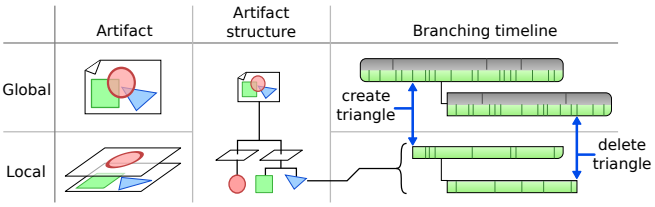

Figure 6. A vector document (left) and its structure (center), and their corresponding history (right). The local history of an element of the artifact (bottom right) is composed of the subset of the commands that affected it (from creation to deletion, branch-wise) and of its own versions along history. Command (triangles) are not represented for readability.

CAUSALITY considers these elements as the set of their versions within the timeline, as shown in Fig. 7. A user command that modifies an element effectively 'ripples' up the artifact or context structure. For example, changing the 'border color' attribute of a vector shape generates a new version of that attribute, as well as a new version of the shape, then recursively of everything that contains it, e.g. groups and layers, up to the artifact itself.

The Command objects in CAUSALITY use *References* to these data objects rather than copies of the objects themselves, as in previous approaches [20]. Importantly, *References* can be one of two types, either *Versions* or *Subscriptions*, as illustrated in Fig. 7.

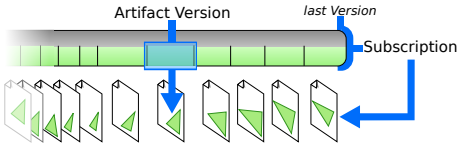

Figure 7. Local history of a triangle shape in a vector document. Versions are fixed in history while Subscriptions 'follow' the corresponding element to the last state of a branch.

A *Version* is a fixed reference in history: it represents a snapshot E_S of an element E at a given state S of history. Later modifications of E generate new Versions $\{E_{i,i\neq S}\}$ but do not alter the commands referencing E_S. However, going back in time to change E_S may affect the commands' outcome.

A *Subscription*, in contrast, is a dynamic link to the last Version of an element in a given branch, regardless of when it was referenced in real time or history. For example, the 'clone' function in Inkscape creates a duplicate T' of a shape T that will mirror every change applied to T: T' is a Subscription to T. Formally, a Subscription E^B echoes every modification of an element E within the branch B. If E is deleted at some state D in this branch, E^B references the last Version E_D of E in that branch. If branching occurs in a branch containing Subscriptions, the user needs to specify which of the original or the replicated branch the Subscription should follow, or duplicate the Subscription.

Since the branching chronology is modeled as a tree, the full sequence of commands and intermediate states that led to a given version of the artifact (or of one of its elements) can be retrieved (Fig. 6), enabling local histories to be exported [15], or represented, as in Chronicle [12] or Chimera [14]. Representations of the artifact can be attached to Versions and Subscriptions as well. Classic editor views typically *subscribe* to the whole artifact: they update their content to its last version within a given branch. Attaching several Subscriptions to a single view enables quick comparison of design alternatives, as in Parallel Paths [27]. Versions provide persistent views of specific artifact states that can be used as visual references, or to observe the effects of a set of modifications on the artifact through probes [12] or with flickering [8].

Commands

CAUSALITY links data describing commands to References of the elements that they influence or use. Every element in the context or artifact is thus linked to its 'causes', i.e. the commands that affected it and the parameters of these commands. CAUSALITY defines commands as septuples:

$\text{LABEL}_{\text{String}}$ identifies the command, similarly to Amulet's `LABEL` slot. It can be format-specific, e.g. 'Create ellipse' in any vector graphics, or application-specific, e.g. a Photoshop filter. The Label is meant to be interpreted by applications or systems, e.g. to re-run past histories or export histories across applications; it cannot be null.

$\text{USER}_{\text{String}}$ identifies the user that triggered the command. $\text{DATE}_{\text{long}}$ is the time(-stamp) of the command in real time.

$\text{TARGETS}_{\{\text{String, Type, Version+}\}*}$ contains the elements to which the command is applied or that will be modified by it. This is similar to Amulet's `OBJECT_MODIFIED` slot except (a) these elements are referenced as Versions rather than as elements, so their previous values can be accessed through history and do not need to be stored in the command, and (b) elements of the context can be referenced as Targets as well. For example, commands that change the application's primary and secondary colors can be accessed, as in Chronicle [12]; however, these commands are linked to the elements they modify through the their Target's set, while in [12] the user has to find a color change in the Color track and find the corresponding event in the Tools track. Targets cannot be Subscriptions since commands are applied to definite history states. Targets can be empty if no element is altered nor created by the command, e.g. 'Save'.

$\text{INPUTS}_{\{\text{String, Type, Reference}\}*}$ records the user input(s) that were used to compute the effect of the command, e.g. a drawing stroke or a set of values entered through a dialog window. Temporal inputs such as strokes can be used to replay commands in real time, as with [8, 12]. The Inputs set only contains the user inputs that were used for computing the command's effects: CAUSALITY filters out less relevant user actions and periods of time, e.g. random cursor movements or pointing to a button before clicking it. Storing these input movements is also much lighter than whole video streams. Inputs are References, and they can be empty.

$\text{PARAMETERS}_{\{\text{String, Type, Reference+}\}*}$ contain every Reference (Versions or Subscriptions) to artifact and context elements that were used to compute the effect of the command, e.g. the last Version of the clipboard for a 'paste' command or the current brush attribute for a 'paint' command. Parameters, in combination with Inputs, should enable the command to be re-run exactly as originally performed. Using Subscriptions as parameters enables higher-level interactions such as Inkscape's cloning. Parameters can be empty as well.

$\text{RESULTS}_{\{\text{String, Type, Version+}\}*}$ references every new Version that the command generated, including new state(s) of history and the new Versions of elements that have been created or altered by the command. For example, the Results set of text entry includes the modified range of caret locations, and the Results set of a paint stroke includes the pixel region that was modified by it. Similarly to Targets, the Results set is bound to the newly generated state(s) of history and thus cannot contain Subscriptions; it can also be empty if no element was modified nor created by the command, e.g. deleting elements

in the artifact. The Results and Targets sets need not be equal; e.g., inserting text Targets a caret *location* and Results in a new character *range*.

Targets, Inputs, Parameters and Results are named and typed (respectively 'String' and 'Type' components). *Names* identify command attributes for later access, modification or reuse. For example, a 3D scaling operation can require three different scaling factors (x, y and z) as parameters. Modifying one of these parameters requires knowing which value corresponds to which factor. *Types* enable coherent reuse of existing command parameters. They can be structured, similarly to Object-Oriented programming; for example, a 'pen stroke' in Photoshop consists of the shape, velocity- and pressure profiles of the actual stroke; themselves are composed of temporal data points. CAUSALITY is 'weakly typed': elements of compatible Types can be swapped, e.g. Integer to Float or ARGB color to HSV color.

Each element of Targets, Parameters and Results may consist of more than one Reference (hence the '+'). For example, one could imagine a vector graphics function that subtracts a set of shapes S from a single shape s and then deletes the shapes in S. Being all modified by the command, s and all the shapes in S are Targets, however they have different roles: the elements of S are considered indistinctly as a group.

Note that this septuple contains the minimal requirements of CAUSALITY and can be augmented for the needs of specific functionalities or applications, e.g. for command filtering based on application-dependent command types [9, 12, 24] or hierarchies of commands [9, 20, 28].

Interferences and Paradoxes

CAUSALITY's branching chronology permits the creation of parallel histories that differ from the linear timeline experienced by the user. Modifications of history generate new branches by *replicating* sets of commands (Fig. 5-right), which causes the generation of a new set of artifact and context Versions. The linear timeline, however, will not contain the replicated commands; only the triggering action executed by the user. Any explicit user command that causes a branch to be created (thus replicating commands) is call an *interference*. Although replicated commands are not included in the linear timeline (Fig. 5), they do appear in the Results set of the interference that generated them, with an updated Date attribute.

By changing courses of actions in the replicated branch, interferences can generate *paradoxes*, i.e. causal conflicts that prevent deterministic computation of further history states. A command is deemed "paradoxical" if at least one of the references necessary to the computation of its effects has been altered or deleted in its (chronological) past; its resulting state(s) cannot be rendered until the paradox is resolved. CAUSALITY defines two types of paradoxes:

Inconsistencies are paradoxes that prevent commands from being run. They occur when at least one element of a command's Parameters or Targets sets no longer exists in the history state from which it is referenced, e.g. changing the font size of a paragraph whose creation has been undone.

Ambiguities are paradoxes that result in multiple possible outcomes. They occur when at least one element of a command's Parameters or Targets sets has changed in the history state from which it was originally referenced. For example, if a shape with a color C1 is copy-pasted at some point in history, and if the user goes back in time to alter the original shape's color ($\rightarrow$ C2) before the copy-paste operation, then there is an uncertainty about whether the pasted shape's color should be the 'original' one from before the interference (C1), or the 'chronological' one that it would have had if the copy-paste command had been triggered in this new branch (C2).

Paradoxes are transitive: every command that targets or uses parts of the Results set of a paradoxical command is also paradoxical. The earlier a paradox is resolved chronologically, the fewer paradoxical commands and states have to be resolved by the user. Paradox resolution can also be deferred. For example, if a user goes back in time to insert several commands one after the other, she should not need to decide what resolution method to use until the whole process is complete, i.e. after the last command is inserted.

As discussed earlier, there can be several sensible ways to resolve a given paradox. CAUSALITY's structure of commands enables a variety of resolution methods adapted from the literature without deleting history:
- *Remove* the command from the replicated subtree, similar to Cascading Undo [6]. The new subtree is created as if the paradoxical command never occurred. Every further command of that subtree that references elements from the Results set of the removed command will be marked paradoxical.
- *Recompute* the outcome of the modified portion of history regardless of its original branch, similar to the Replay operation in [4]; this is only possible with Ambiguities, for Inconsistencies cannot be run by definition.
- *Ignore* the paradox by reusing the Results of the original command in the replicated branch. This can be compared to the default behavior of Amulet [20], except these elements are referenced, not copied: if an interference modifies these Results later in real time, the user will have the possibility to reflect these changes in the Ignored command as well.
- *Substitute* the altered or missing Targets or Parameters with compatible replacements chosen by the user, as suggested in [15,17], in the form of References from other history states.

Resolution methods do not generate new subtrees, as opposed to interferences: their goal is to 'fix' newly replicated subtrees. They only appear in the user timeline and, as for other commands, they can be accessed and modified later.

EXAMPLES OF USE IN IMAGE EDITING

In this section we use a painting scenario to illustrate several novel, powerful and exciting interaction possibilities that are made available by modeling history with CAUSALITY. As the focus of this paper is on the underlying model, we describe functionalities without considering how that functionality might be accessed and controlled through interactive components or visualizations. Thus, the interfaces in the figures are only illustrative and should not be considered for their usability or visual design.

The basic scenario concerns a user editing a scene image with digital painting software that models history using CAUSALITY. The scenario starts with the painting in the state shown Fig. 8-left, with everything painted on a single layer. The river on the right was painted most recently. The user wants to change the painting as follows: the house should be bigger (but not the boat next to it), and another similar house should be shown partially occluded by the original one (Fig. 8-right).

Figure 8. Original image and target image after modifications.

One forward method to perform these changes is to (a) select the pixels of the house and copy them, (b) scale these pixels to make the house bigger, (c) create a new layer with that bigger selection as a negative mask, so that the first house remains in the foreground, (d) paste the copied pixels in the new layer, (e) scale them down to make the second house look smaller, then (f) flatten the image. However this method raises a number of issues: first, it only works with applications that support layers (c, d); second, it requires a non-novice knowledge of selection tools (a) and of the way layers and masks work (c, d, f); third, scaling pixels up or down (b, e) results in regions of the image with visibly different resolutions; fourth, if the user wants to change the size of the foreground house again, he will have to repeat the whole process.

CAUSALITY allows the user to alter his own past actions: he can change the *history* of the painting as if these changes had been made from the start and without needing layers or complex selections. We will illustrate the following process: (a) select the *commands* that correspond to painting the house, (b) scale up the corresponding *strokes*, i.e. the user's input when he painted the house and the paint brush sizes, (c) copy these commands and apply them again in their own past so that the original house is drawn on top of them, then (d) scale their stroke inputs and brush sizes down as in step (b).

(a) Selecting commands

We need to select the commands that were used to paint the house in order to modify them. To do so, we will use the various components of a CAUSALITY command: we will filter them by their name, the location of their effect, the time they occurred and their color parameter.

First, we are only interested in painting commands, so we can use this Label as a first filter. Second, the Results set of a 'paint' command references the pixel region that it affected; conversely, any pixel region of the document is linked to the commands that affected it. The commands that were used in the global vicinity of the house can thus be accessed from a simple selection operation (Fig. 9-left). The selected commands can be displayed on a timeline[2] thanks to their Date attribute (bottom panels in Fig. 9).

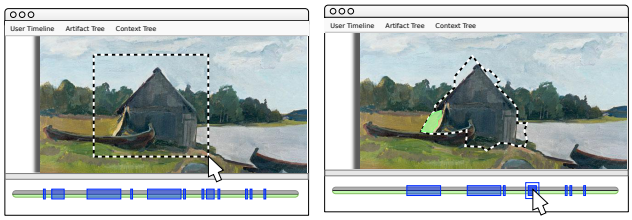

Figure 9. Left: selecting pixels on the image highlights the commands that affected them and the time these command occurred (bottom panel, blue). Right: hovering the timeline with the cursor highlights the pixels resulting from the corresponding command (top panel, green) and has been used to unselect commands.

Third, the selected commands form clusters on the timeline: most of the house-painting commands were performed at approximately the same time, while the ones corresponding to the sky, the grass, the trees, the boat and the river inside the selection rectangle were performed at different times. Similar to pixels, periods of time are linked to the commands that were invoked within them: one can thus unselect commands by toggling their corresponding time region on the timeline. To facilitate this process, the interface could highlight the pixel region affected by the currently hovered command (green in Fig. 9-right).

Filtering commands by name, location and time provides fast and coarse selection, however some unwanted commands may remain selected if they occurred close to the house and approximately at the same time of the house-painting commands, e.g. sky-colored paint strokes added near the top of the roof for edge adjustment. The Parameters set of a 'paint' command references the application's 'primary color' attribute at the time of the command, the user can thus recover the list of colors that were used by the selected painting commands (Fig. 10-left) and use it to unselect the remaining non-house-painting commands (Fig. 10-right).

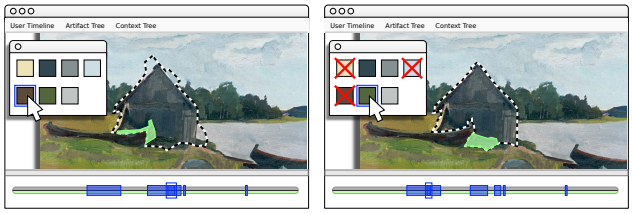

Figure 10. Left: the colors used in the selected set of commands can be displayed (foreground window); hovering them highlights the corresponding pixels and commands in the top and bottom panels. Right: removing colors from this set unselects the corresponding commands.

(b) Scaling paint strokes

The Inputs set of a 'paint' command contains the user's stroke in the form of a set of cursor locations, and possibly a velocity and pressure profile. Its Parameters set also refers to the brush that it used to compute its effect. The user can apply any transformation on this data; in this case, he will scale both the

[2]In this section we will only show periods of time (rather than actual commands) for the sake of readability.

strokes and the brush widths (Fig. 11, right) and recompute the artifact's history to observe its effects.

This is a modification of history: a parallel branch is generated (bottom panel in Fig. 11-right), branching from the last Version of the artifact that is common to both branches. These modifications cause Ambiguity paradoxes to be raised: later 'paint' commands have their resulting pixel regions intersecting the ones of the modified commands (yellow in Fig. 11-right). Some of these Ambiguous commands must be deleted from the new branch, e.g. overlapping landscape-painting commands that were performed after the house was painted (chronologically). As a result, the modified house looks as if it was drawn this big from the start, without any loss of resolution. This is an interesting application of history: modifying inputs provides vector-like functionalities to tools intended to manipulate raw data (e.g. pixels).

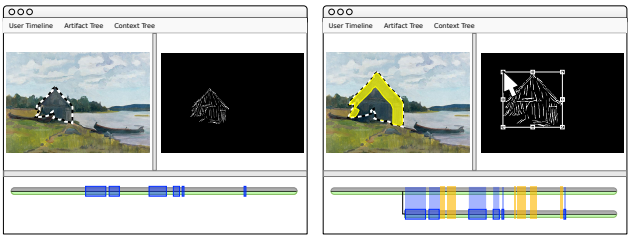

Figure 11. Left: displaying the stroke shapes corresponding to the selected 'paint' commands (right panel). Right: scaling these strokes generates a parallel branch (bottom panel) and raises paradoxes (in yellow) with later, intersecting 'paint' commands.

(c, d) Reusing commands in the 'past'

CAUSALITY considers commands as Versions, similarly to artifact and context elements, that can thus be referenced and reused as well. The commands used to draw the house can be copied as a group, even if they weren't all contiguous in time, and reapplied anywhere in history (Fig. 12-top).

Remember however that the color and brush Parameters of a 'paint' command are References: duplicating a paint command somewhere in time means to either perform a similar painting *stroke* with the brush and primary color of the history state at which they are replicated, or to reproduce the actual paint operation including color and brush. In the former case, the brush and color Parameters reference the Version of the application's brush and primary color attributes where the commands were applied. In the latter, they reference the same Version of these attributes as the original commands.

In our case, the user duplicates these commands using the original parameters and *before* the ones corresponding to the original, now bigger house (Fig. 12-bottom); these new commands thus happen earlier in chronology, so their result will be partly hidden by the bigger house. This will again generate a new subtree with some Ambiguous commands (Fig. 12-bottom). Finally, the stroke Inputs of these commands can be scaled down using the same process than in the previous step.

Discussion

This scenario illustrates how CAUSALITY enables safe, flexible and understandable modifications of history that are substantially more powerful than existing history mechanisms.

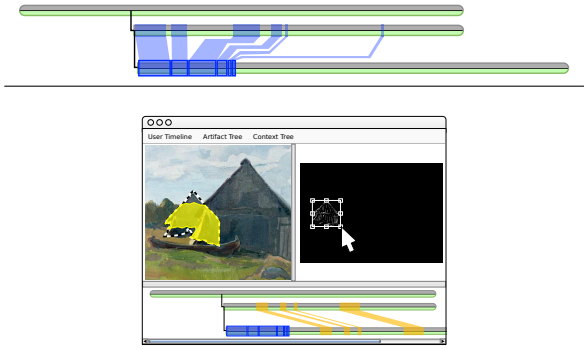

Figure 12. Top: re-applying all the selected commands in their own past generates a new parallel branch; the replicated commands are linked to the original ones. Bottom: after relocation and scaling, these commands generate new paradoxes.

Commands can be modified, duplicated and reordered: the user can recover from earlier errors or rethink the document without losing any previous work. Further, it reduces the need for users to form a strategy for completing their work before beginning to work with the interface.

The functionalities described in this scenario are but a glimpse of the possibilities that CAUSALITY can offer. Our goal is not to replace advanced editing but to increase the vocabulary of corrective and editing operations available to interaction designers, and ultimately to the user. In our example, modifying the construction of the scene by classic means requires some training in layers, masks and selection techniques. It also implies local changes of resolution and erasing existing parts of the landscape. CAUSALITY enables a process-oriented approach where users can simply re-access, manipulate and alter the operations they know they did without erasing any previous actions or result.

Our scenario used a graphical application because it is easy to describe and illustrate. As a conceptual model, CAUSALITY can be applied to any kind of editing software, from basic text editors to video and audio editing software or multi-file project managers. It is however challenging to define a universal implementation framework from it, for different application domains will likely have very different performance and data requirements. For example, systematically storing all the states of a multi-layer graphic document might become very voluminous, while some of them could be made *accessible* through recomputation without being *stored*. Additionally, fully integrating CAUSALITY into pre-existing software systems would require information that is seldom available from software APIs, necessitating access to source code.

CONCLUSION AND FUTURE WORKS

This paper introduced CAUSALITY, a new conceptual model of interaction history that keeps track of past commands and states of the edited artifact in the form of a causal system. CAUSALITY can model most of the existing systems and techniques of the literature and it enables more flexible opportunities for temporal interactions. It can also provide a richer vocabulary of effects and consequences. As well as exactly stipulating components of CAUSALITY, as necessary

for its deployment, we also provided examples of its use in an advanced painting application scenario.

Our next step will be to implement and evaluate CAUSALITY in usable applications, focusing on the following two points. First, we will implement CAUSALITY both as the core of a specific application for evaluation, and as a plugin of a scriptable application such as Adobe Photoshop or Eclipse; the latter will help us evaluate how far we can go with CAUSALITY's functionalities when using APIs. Second, we will design and evaluate user interfaces that provide access to the power of the novel interactions inspired by CAUSALITY. Visualizations, such as those suggested in our painting scenario, are likely to have a major impact on the effectiveness of the user's interaction. As illustrated by the scenario, CAUSALITY permits the invention of new interactive techniques for manipulating the history of a document based on reuse and modification of the links between interaction causes and effects. Ultimately, we hope that CAUSALITY will inspire others to invent, implement and evaluate new, learnable and powerful tools for interacting with time.

REFERENCES

1. Abowd, G. D., and Dix, A. J. Giving undo attention. *Interact. Comput. 4*, 3 (Dec. 1992), 317–342.
2. Appert, C., Chapuis, O., and Pietriga, E. Dwell-and-spring: undo for direct manipulation. CHI '12 (2012), 1957–1966.
3. Archer, Jr., J. E., Conway, R., and Schneider, F. B. User recovery and reversal in interactive systems. *ACM Trans. Program. Lang. Syst. 6*, 1 (Jan. 1984), 1–19.
4. Brown, A. B., and Patterson, D. A. Rewind, repair, replay: three r's to dependability. SIGOPS European Workshop 10 (2002), 70–77.
5. Bueno, C., Crossland, S., Lutteroth, C., and Weber, G. Rewriting history: more power to creative people. OzCHI '11 (2011), 62–71.
6. Cass, A. G., and Fernandes, C. S. T. Using task models for cascading selective undo. TAMODIA'06 (2007), 186–201.
7. Cass, A. G., Fernandes, C. S. T., and Polidore, A. An empirical evaluation of undo mechanisms. NordiCHI '06 (2006), 19–27.
8. Chen, H.-T., Wei, L.-Y., and Chang, C.-F. Nonlinear revision control for images. SIGGRAPH '11 (2011), 105:1–105:10.
9. Edwards, W. K., Igarashi, T., LaMarca, A., and Mynatt, E. D. A temporal model for multi-level undo and redo. UIST '00 (2000), 31–40.
10. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1995.
11. Greenberg, S., and Witten, I. H. Supporting command reuse: empirical foundations and principles. *Int. J. Man-Mach. Stud. 39*, 3 (Sept. 1993), 353–390.
12. Grossman, T., Matejka, J., and Fitzmaurice, G. Chronicle: capture, exploration, and playback of document workflow histories. UIST '10 (2010), 143–152.
13. Heer, J., Mackinlay, J., Stolte, C., and Agrawala, M. Graphical histories for visualization: Supporting analysis, communication, and evaluation. *IEEE TVCG 14*, 6 (Nov. 2008), 1189–1196.
14. Kurlander, D., and Feiner, S. A visual language for browsing, undoing, and redoing graphical interface commands. In *Visual Languages and Visual Programming* (1990), 257–275.
15. Kurlander, D., and Feiner, S. A history-based macro by example system. UIST '92 (1992), 99–106.
16. Loregian, M. Undo for mobile phones: does your mobile phone need an undo key? do you? NordiCHI '08 (2008), 274–282.
17. Loregian, M., and Locatelli, M. P. An experimental analysis of undo in ubiquitous computing environments. UIC '08 (2008), 505–519.
18. Lunzer, A., and Hornbæk, K. Subjunctive interfaces: Extending applications to support parallel setup, viewing and control of alternative scenarios. *ACM Trans. Comput.-Hum. Interact. 14*, 4 (Jan. 2008), 17:1–17:44.
19. Martin, R., Demme, J., and Sethumadhavan, S. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *SIGARCH Comput. Archit. News 40*, 3 (June 2012), 118–129.
20. Myers, B. A., and Kosbie, D. S. Reusable hierarchical command objects. CHI '96 (1996), 260–267.
21. Naderlinger, A., and Templ, J. A framework for command processing in java/swing programs based on the mvc pattern. PPPJ '08 (2008), 35–42.
22. O'Brien, J., and Shapiro, M. Undo for anyone, anywhere, anytime. SIGOPS European Workshop 11 (2004).
23. Prakash, A., and Knister, M. J. A framework for undoing actions in collaborative systems. *ACM Trans. Comput.-Hum. Interact. 1*, 4 (Dec. 1994), 295–330.
24. Rekimoto, J. Time-machine computing: a time-centric approach for the information environment. UIST '99 (1999), 45–54.
25. Seifried, T., Rendl, C., Haller, M., and Scott, S. Regional undo/redo techniques for large interactive surfaces. CHI '12 (2012), 2855–2864.
26. Shneiderman, B. Direct manipulation: A step beyond programming languages (abstract only). CHI '81 (1981), 143.
27. Terry, M., Mynatt, E. D., Nakakoji, K., and Yamamoto, Y. Variation in element and action: supporting simultaneous development of alternative solutions. CHI '04 (2004), 711–718.
28. Washizaki, H., and Fukazawa, Y. Dynamic hierarchical undo facility in a fine-grained component environment. CRPIT '02 (2002), 191–199.