Rewriting History: More Power to Creative People

Carlo Bueno, Sarah Crossland
School of Engineering
University of Auckland
{cbue001, scro089}@aucklanduni.ac.nz

Christof Lutteroth, Gerald Weber
Department of Computer Science
University of Auckland
{christof, gerald}@cs.auckland.ac.nz

ABSTRACT
Trying out different alternatives is a natural part of creative work, resulting in several versions that are hard to manage. With the tools available today, we often end up having to manually redo changes that worked in one version on other versions. We propose a new approach for supporting creative work: an artifact is described as the history of the operations that created it. We show that by allowing users to change this history, the common use cases of merging, generalizing and specializing can be supported efficiently. This *rewriting history* approach is based on a formal specification of the operations offered by a tool, leads to a new theory of operations, and enables exciting new ways to share and combine creative work. It is complementary to state-based version control, and offers the user a new understanding of merging. The approach was implemented for a collaborative drawing tool, and evaluated in a user study. The study shows that users understand the approach and would like to use it in their own creative work.

Author Keywords
Interaction framework, creative work, collaboration

ACM Classification Keywords
H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

INTRODUCTION
Creating artifacts such as diagrams is an essential part of many people's work. As the work on such artifacts progresses, many versions are created. This can happen in the process of creating a single deliverable version, or if artifacts are reused on many different occasions, requiring slight modifications. For example, people may use presentation slides for different talks, changing some of them each time. These situations lead to inefficiencies, since useful work on one of the versions cannot be reused in other versions in a straightforward manner. The following simple story highlights the problem:

Ann and Bob are designing a new company logo. They start off with a simple circle. Ann makes a copy for herself so that she can work independently of Bob on the color scheme. In market research she finds out that "green is the new black" and changes the circle's fill color to green. Meanwhile Bob resizes the circle into an ellipse so that it matches retro designs currently in vogue. The history of their changes and result of their work can be seen in Figure 1.

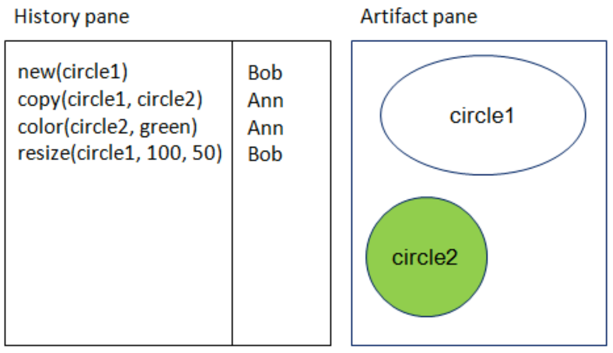


Figure 1. Collaborative editing of a simple diagram

Ann and Bob are convinced that they did hard work on their versions of the logo and do not want to redo anything. Hence they want to merge their versions into a single version of the logo using a tool. Both are disappointed to find that none of their drawing tools can do the merge. Ann says, "I should have done the change of the color after you did your changes." Bob replies "but then we wouldn't be done by now either." Ann and Bob now face the prospect of merging the diagrams by hand, which is tricky and time consuming because it means that one of them has to redo all work on the other's version.

The story illustrates a common limitation of creative tools: once a user has performed a sequence of operations on an artifact such as a logo or a diagram, the information about that sequence – the *history* – is mostly lost. Many editors have undo and redo functions that allow users to go back and forth between previous states of the artifact. But the fact remains: such functions only allow users to access *states* of the artifact, but not information about the *operations* that make up its history. Some problems such as the merging problem in the story could be solved if the history of an artifact could be changed. For example, Ann's wish to perform her changes after Bob's could be granted in retrospect.

In this paper, we propose history rewriting as an approach that can be used to satisfy common use cases of creative work. As we can see from the story, being able to edit the history of an artifact in retrospect would give users more freedom in their design. Decisions which were made earlier on in the creative process would be no longer painful and time consuming to change. This leads to a new theory of history rewriting for creative work, and gives rise to questions about how the approach can be made useful for end-users.

This paper is structured as follows: first, we look at the motivation for the rewriting history method, and then at related work. Afterwards we explain the concepts of writing and rewriting history. The theory behind some of these ideas is explained next, followed by an evaluation that investigates how easily the rewriting history approach can be applied and understood. The paper concludes with an overview of future work and a summary of important contributions.

USE CASES

The history of an artifact consists of all the operations that were performed on it. In the rewriting history approach, this history can be edited in hindsight, resulting in a *counterfactual history* and generally a different artifact. Common use cases that can be addressed with history rewriting occur in the creative process if we have found the right operations, but not applied them on all the right objects. Without history rewriting, this would mean that we have to redo work. This can happen in any of the following three use cases:

Merging means that we have different versions A and B of the same objects. We want to have only one version that combines the operations that we applied to create versions A and B. An example of this is the story in the introduction.

Generalizing means that changes need to be applied more generally, i.e. to a superset of the objects to which they were applied originally. For example, after changing the color of one of the circles in a diagram, we may want to change all the circles in the diagram to the same color. Merging can be reduced to generalization.

Specializing means that changes need to be applied more specifically, i.e. to just a subset of the objects to which they were applied originally. For example, after setting the color of all circles in a diagram to red, we may decide that only some of the circles should have that color. Specializing is the inverse of generalizing.

These use cases can already occur if a single user works on a single document that evolves over time and is used in different contexts in slightly different versions. Presentation slide sets are a typical example. However, these use cases can also occur during collaborations. The more people are involved, the greater is the integration effort required at the end when the individual contributions are given their place in the overall work.

If the above use cases are not supported by a tool, the amount of work that needs to be redone grows with the size of the creative work. There are tasks that require several modifications without history rewriting, but only one modification with history rewriting. For example, imagine a shape was copied several times and the copies arranged in a pattern. If we wanted to change the color of all the copies without history rewriting, we would have to change them individually. With history rewriting, we could simply change the color of the original shape before it was copied.

RELATED WORK

Most applications for creative work record the *factual history* of an artifact, i.e. they keep track of artifact operations. The factual history is mostly used for undoing and redoing, and typically only the most recent operations can be undone or redone. The history operations that can be used to create a counterfactual history are very limited.

Extensive research has been done on the visualization of operation histories (Nakamura et al., 2008), and the possibility of using it for documentation and learning (Grossman et al., 2010). Branching has been considered as well, but only as a way to remember operations that have been undone (Heer et al., 2008), and not as a tool to manage variations of artifacts. The possibility of editing the operation history has been proposed (Kurlander et al., 1988), but its semantics, use cases and benefits have not been explored. In particular, the implications of reordering and its uses have not been considered before.

On the surface, the rewriting history approach looks similar to version control systems (VCS) such as SVN, Git and Hg. Such VCSs offer powerful functions for managing and merging different versions of files, offering support for similar use cases. However, there are significant differences:

1) The abovementioned VCSs *are state-based*, as opposed to our approach, which is *operation-based*. That is, they do not record artifact operations but merely compare two states of an artifact. The differences are recorded on a lower level of abstraction, as insertions and deletions on the raw artifact data. There are operation-based VCSs that are integrated with editors, but they are highly domain specific and do not support history rewriting (Koegel et al., 2010).

2) The abovementioned VCS are *unstructured*, i.e. they are not aware of the syntax of an artifact, but merely consider changes between states on a lexical level. For example, they are not aware that the color of an object was changed, but merely see that one string was replaced by another. This leads to merging conflicts (Mens, 2002) that can only be avoided with *syntactic merging*, i.e. by taking into account the syntax of an artifact (Conradi et al., 1998). *Structured* VCSs have knowledge about the artifact syntax. However, they are typically state-based and consider only the syntax of the artifacts themselves, but not the syntax of their history operations. For example, the Pounamu diagram editor (Mehra et al., 2005) has a structured model of diagrams, and can compare and merge diagrams according to this model. But it has no knowledge of the history of a diagram, as this is not part of the model. Similar state-based syntactic merge tools exist for source code of various programming languages (Hashimoto et al., 2008; Apiwattenapong et al., 2007), and graph-like object structures in general (Zündorf et al., 2009).

3) VCSs such as Git allow power users some rewriting of the version history, but this is difficult and situated on a lower level of abstraction, as described in the previous points. It also creates problems for collaboration: if changes have already been pushed to a central server and

are then rewritten and pushed again, this creates an alternate version of those changes on the central server (Scott, 2009). Collaborators will be confused as it becomes unclear which version they should base their own work on.

In summary, VCSs are no substitute for the rewriting history approach, but are complementary: a structured VCS could be used on a lower level to manage the *complete history* that includes artifact operations as well as history operations. This would make it possible, for example, to undo and redo history operations.

Operational transformation (OT) (Ellis et al., 1989; Agustina et al., 2008) is a popular technique for managing concurrency in systems for synchronous collaborative work, by exchanging and transforming the history of user operations. If operations are executed concurrently by different collaborators, they are transformed upon reception so that each collaborator sees the same consistent state. However, the approach is orthogonal to our approach: our approach provides added value even for a single user, while OT is only needed for synchronous collaboration. In OT the history of the operations cannot be changed by the collaborators. It may be discarded once a state of consistency has been reached.

Although some of the previous work has addressed some of the issues discussed here, there is no previous publication that does all of the following: developing an algebraic model of rewriting a history of operations, applying this model to common use cases in creative work, presenting a streamlined tool that implements this model, and performing a user study that evaluates whether the model is understandable.

WRITING HISTORY

Traditionally, a document stores only the result of creative work, and not its history. In our approach, a document describing a creative artifact contains primarily a history of *artifact operations*. An artifact is defined through re-execution of the history because all artifact operations are deterministic. In an almost literal translation of this view, a tool implementing our approach has two presentation panes: the *history pane* and the *artifact pane*. An example of this is shown in Figure 1.

A tool keeps track of all artifact operations executed by the user and stores them in a list. In this article we focus exclusively on a drawing application for the sake of brevity, but the approach can be applied to a large number of WYSIWYG tools. Each operation has an object that it applies to and an arbitrary number of parameters, for example x-y coordinates from point-and-click operations. Initially the operations are ordered by their execution time.

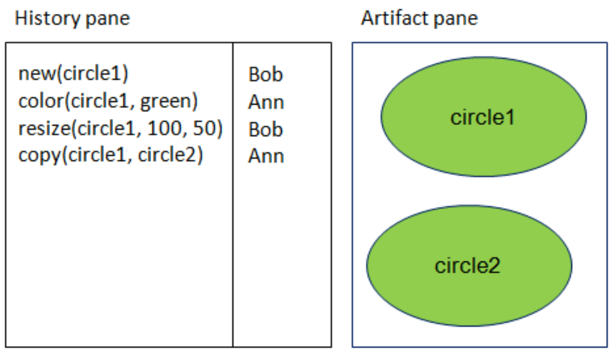
REWRITING HISTORY

Since the history is stored in the data model, it is now possible to change the history using new kinds of operations, *history operations*. It is possible to change the artifact by rewriting the history. The set of history operations needed is very small: first, a swap of two consecutive artifact operations; secondly, a deletion of artifact operations in the current version of the history. As a result of many swaps, the order of the history can be arbitrarily changed. History operations are invoked in the history pane of the user interface.

Many operations in classic drawing applications can have different semantics. The different semantics have different consequences with regard to swapping. For example, a move of an object can be interpreted as a move to an absolute position, or alternatively as a move to a position relative to the old position. With respect to swapping, semantics which are *commutative* behave differently than semantics that are *non-commutative*. An absolute move overwrites the object position and is therefore a non-commutative operation, i.e. swapping two absolute moves may change the artifact. A relative move can be defined as an addition of an offset, and is a commutative operation due to the commutativity of vector addition. Hence, swapping two relative moves does not change the artifact. In general, swapping commutative operations does not change the artifact, while swapping non-commutative operations may change the artifact.

An important non-commutative history operation that requires special attention is copy. Semantically, on the low level of object identities, the copy operation is asymmetric: the original object is kept unchanged, and a distinct clone is created. For the original, the copy is semantically a skip (a non-operation), but not for the clone. This has to be taken into account in swaps that involve a copy operation. There are two such swaps.

First, let us consider a swap that moves an operation *e* after the copy to the position before the copy. We consider two cases. If *e* is an operation on the original, then the swap has no consequence for the original. But for the clone the swap has the effect that *e* is now also applied to the clone. This is the case with the resize operation in Figure 2, which shows how Bob's wish from the motivation section can be fulfilled by moving the copy after the resize. This operation can solve the use case of **generalizing**, and therefore we use this term for such a swap.



new(circle1)	Bob
color(circle1, green)	Ann
resize(circle1, 100, 50)	Bob
copy(circle1, circle2)	Ann

Figure 2. History of merged versions

If *e* is an operation on the clone, then *e* is at first undefined before the copy because the clone does not exist yet. Therefore we extend the semantics of swap: if the operation *e* on the clone is moved before the copy operation, then *e* is applied to the original (i.e. its ancestor) instead of to the clone.

The second kind of swap involves moving an operation *e* to a position after a copy. If *e* is invoked on the object that is copied, then the user can interactively specify whether the operation should be applied to the original object or to the clone. This operation can solve the use case of **specializing**. As mentioned before, specializing is the inverse of generalizing. Consequently, the corresponding changes on the history are also inverse to one another (moving operations up or down).

Merging of different versions with a common ancestor, as in the second example of Ann and Bob, is a combination of several generalizing operations. The operations that were invoked on the copies are moved to positions before the copy operations. This is illustrated in Figure 2.

The other history operation is the delete operation. This operation allows the user to remove an artifact operation from the history. There exist *dependencies* between operations. Some operations are responsible for creating new artifacts and every operation which then works on that artifact is then said to be dependent upon the operation which created it. When this creation operation is deleted, each dependent operation is deleted as well. This is because the dependent operations are now unnecessary, as they no longer affect any shape in the diagram. This feature is called cascading delete.

The name cascading delete comes from the fact that this delete is propagated down the history, i.e. forward in time. As with the swap operation, the copy operation presents a special case. We cannot just delete all the dependent copy operations. We also need to delete all other operations which are dependent upon that copy operation. This effect means that it is possible to clear all artifacts, and all operations, from the history by deleting just one operation, if the whole history is dependent upon that one operation.

THEORY

In order to give precise semantics to the history operations, we also have to give precise semantics to the artifact operations. We use an algebraic model for the editor and its artifact operations. Operations are modeled as functions on the artifact. Executing operations m1, c2 after each other is therefore mapped to function concatenation m1·c2, and this is known to be the fundamental associative operation in set theory. This model is the primary justification, why histories are just sequences and not expression trees: A history (m1 c2)(c3 m4) is the same as m1(c2(c3 m4)) and is always just (m1 c2 c3 m4). Given that all artifact operations are associative, the question arises if they are commutative.

If they would all be commutative, our history rewriting approach would make no difference. Artifact operations are sometimes commutative and sometimes not. Note that commutativity is only defined on neighboring artifact operations. In our tool, we have decided to offer the swap operation by two buttons. It is natural that the user always selects operations in the history. For a selected operation, there are two natural swap operations, up and down. Each of them has its own button. Distinguishing commutative and non-commutative operations enables us to offer an important advanced feature of the history panel, namely skipping of commutative operations, which we will explain now.

Commutativity and its uses

If two neighboring artifact operations are commutative, swapping them does not make a difference. This allows us to create a powerful feature. Pressing the swap button for one artifact operation in one direction will cause that operation to jump a whole set of neighboring artifact operations. One artifact operation will be swapped to the next position where it will produce a change on the diagram. This simplifies the process of changing history.

For understanding if two operations are commutative we first need to look at how an operation is defined. Generally, an operation consists of three parts: a type, a shape, and additional parameters. The type identifies what kind of operation was performed, for example a move or color operation. The shape is the one the operation is applied on. Additional parameters provide information that is specific for the type, such as a new color for color operations. The copy operation is special in that it has a second shape as parameter, which is the new shape produced by the operation.

Commutativity for shape-disjointness

If two operations do not refer to the same shapes, we call them *shape disjoint*. If two operations are shape disjoint, then they are commutative.

All artifact operations are defined to only affect the shapes they refer to. This is no arbitrary choice but is necessary in order to avoid a gulf of execution: if an operation would have effects on shapes it does not explicitly refer to, this would confuse the user as to how the operation could only be applied to the shapes the operation actually refers to. It would also create a gulf of evaluation: the other affected shapes are not listed in the history panel. If two operations apply to different shapes, then their order of execution makes no difference to the final product.

For example, a copy operation would be non-commutative with another operation if that operation acted upon the shape that is copied or the produced new shape. In both cases, the two operations would not be shape disjoint. The same is true for operations that refer only to one shape.

Commutativity for type-disjointness

If two operations have different types, we call them *type disjoint*. For the types of operations that our tool supports, except the copy operation, the following holds: if two operations are type disjoint then they are commutative.

We have defined the operations in our tool so that each operation type affects a different property of a shape, with no overlap between them. Because properties are independent of one another, the order in which different properties of a shape are changed does not affect the outcome.

The copy operation is again a special case. If the copy operation is type disjoint with another operation, they are not necessarily commutative. Swapping them may cause specialization or generalization, as described before. For example, imagine if we had a circle *c1* and a copy operation producing a second circle *c2*. Now we change the color of the first circle, *c1* as shown in Figure 3. If we look at the history of operations, we see that we have two operations which are of different types, but they are actually not commutative as the lemmas above would lead us to believe. This is because initially the color operation only affects *c1*, but if we move it above the copy operation it also affects *c2*.

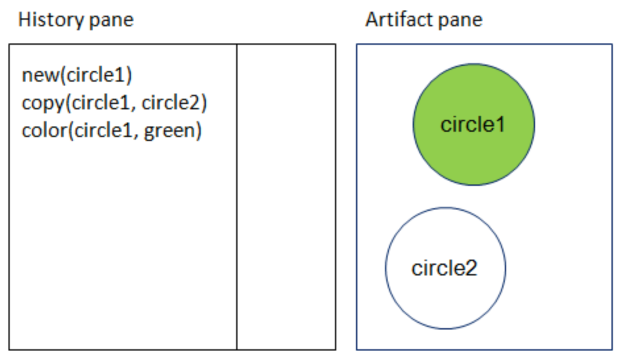


Figure 3. Commutative operations

Commutative neighborhoods

We have now described how to determine if two operations are commutative with one other. However, we would like to expand this to create *commutative neighborhoods*, where several neighboring operations are commutative. It would be helpful if we could identify any property of commutative operations that make it easier to find commutative neighborhoods.

Transitivity is a common mathematical property of relations. In the context of commutativity, transitivity would mean that if an operation A is commutative with operation B and operation B is commutative with C then operation A is also commutative with C. This would be a useful property since it would make the definition and identification of commutative neighborhoods much easier. However, this property does not hold, as proven by the following example. If we have an operation coloring a circle *c1* and a second operation moving a second circle *c2*, then these two operations are commutative. Similarly, the second operation would be commutative with a third operation which is also coloring *c1*. So both the first and second, and second and third operations are commutative, but we can see that the first and third operations are not commutative since they have neither different types nor different shapes. Swapping them results indeed in a different color. This proves that commutativity of operations is not transitive.

It should be noted that we assume that all artifact operations except copy are defined in a form that is called idempotent: Executing them twice has the same effect as executing them once, as in the case with coloring. Similarly, resizing is defined in an idempotent manner, i.e. by defining an absolute resulting size, not a relative size change. Extending the theory to the corresponding relative operations is a further interesting project.

One question when partitioning the history into commutative neighborhoods is: where does a particular commutative neighborhood end? The lack of transitivity has the following consequence: For a given history, one cannot partition the operations into disjoint commutative neighborhoods. Instead, using a different artifact operation as a starting point produces in general a different commutative neighborhood. An example of this is given below.

Assume we have the history shown in Figure 4. We have two new() operations and two other operations referring to the same shape as the first new operation. For the first new() operation, only the two new() operations together are the commutative neighborhood. For the second new() operation, all four operations comprise the commutative neighborhood.

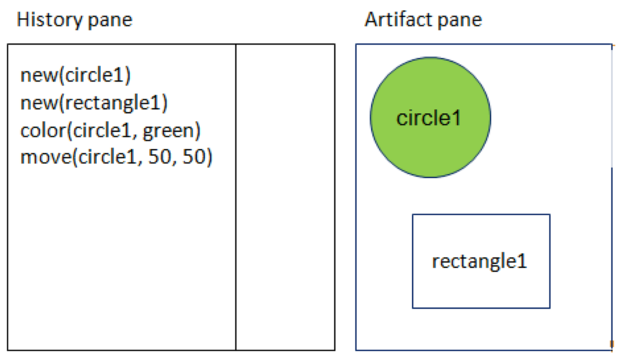


Figure 4. Defining commutative sets

Hence commutative neighborhood should be defined as a function mapping one artifact operation to a set of artifact operations, its neighborhood. This fits well to the aim of defining commutative operations: we want to define commutative neighborhoods in such a way that when we move an operation in the history, we move it to the next position in either direction which would make a change to the artifacts displayed, in effect skipping over the commutative neighborhood. Therefore the selected operation is the starting point for defining the neighborhood, which then stretches both upwards and downwards.

Cascading delete

Cascading delete works on the principle of dependencies. If we delete an operation, all other operations which are dependent upon it are also deleted. This makes sure that only operations affecting the diagram are listed in the history.

To put this into practice, we need a definition of which operations are dependent on which other operations. We look at the shapes referred to in each operation: operations are dependent upon the operation which created the shape they are referring to, which is either a new operation or a copy operation.

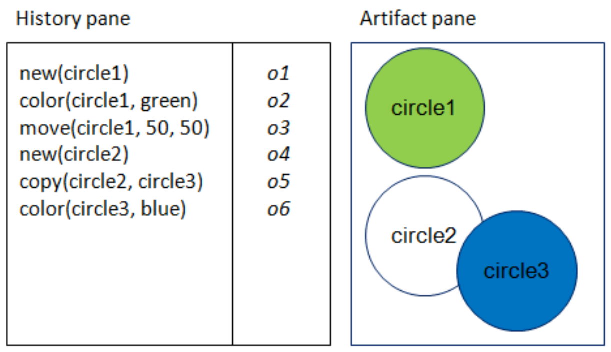


Figure 5. Cascading delete

For example, assume we have three operations, *o1*, *o2* and *o3*. *o1* is a new operation creating a circle, *o2* is a color operation on that circle, and *o3* is a move operation on that circle, as shown in Figure 5. In this example both *o2* and *o3* are dependent upon *o1*, but they are not dependent upon one another.

As we have seen before, the copy operation acts slightly differently from all other operations. Due to the generalizing and specializing features, dependencies on copy operations are not as strong as they are on new operations. If an operation depends on a copy operation, we call this a *soft dependency*. It is soft because the operation is able to be swapped above the copy operation, and as a result become dependent upon the shape which is being cloned in the copy, the original. If an operation depends on a new operation, we call this a *hard dependency*. The operation cannot be moved above the new operation.

For example, consider three operations, *o4*, *o5* and *o6*, where *o4* is a new operation creating a new circle, *o5* is a copy operation on this circle, and *o6* is a color on this copy object, as shown in Figure 5. Initially, *o6* has a soft dependency on *o5*, and *o5* has a hard dependency on *o4*. However, if we now swap the ordering of *o5* and *o6* we have a different set of dependencies. Now we get both *o5* and *o6* having hard dependencies on *o4*.

IMPLEMENTATION

To understand the possibilities that are offered by the concept of rewriting history, we translated our history model one-to-one into a tool. The tool stores its model in a lightweight database system that supports event notification. The event mechanism is used for view maintenance and multi-user support. The implementation supports synchronous collaboration between different users, i.e. changes of one user become immediately visible to other users. However, the aspects of distributed synchronous collaboration are not the main thrust of the theoretical and practical work presented here, but a welcome added benefit. A screenshot of our prototype is shown in Figure 6.

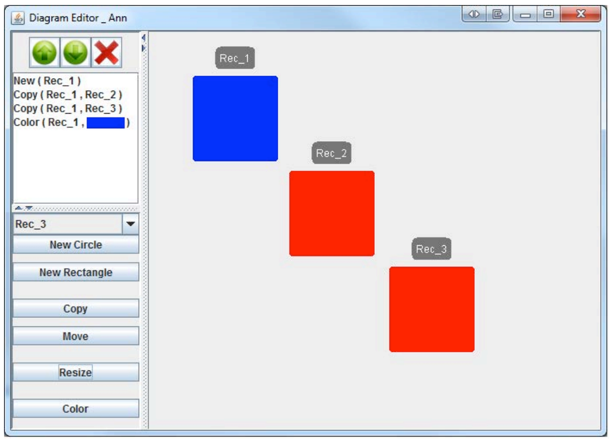
In terms of the model-view-controller pattern, the history and artifact pane are views of the same data model, i.e. the counterfactual history. The two important parts of the tool are the presentation function for view maintenance, and the input control function for model updates. The presentation function can be implemented in a single global refresh routine that simply re-executes the whole history. Using caching this approach can be optimized so that it scales to long histories.

EVALUATION

After the creation of our prototype we performed a usability study. The evaluation itself was not performed on the prototype but as a test of theoretical understanding: The aim of the study was to assess how easily a user could understand and subsequently apply the concept of rewriting history, assuming that the user is familiar with office tools but unfamiliar with our history editor.

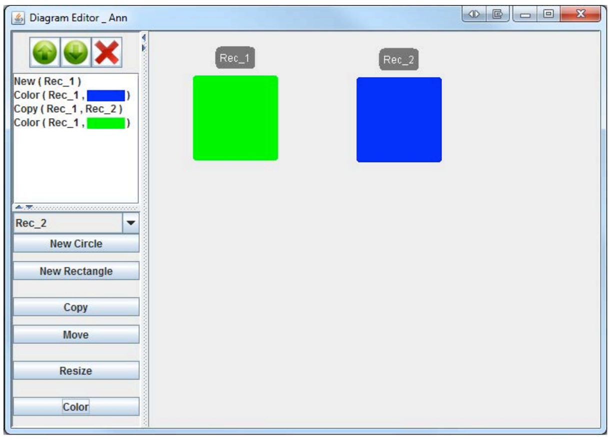
Question (issue evaluated)	Figure	Results
1. How would you change the color of Rec_2 to be the same color as Rec_1? (applying generalization for non-repetitive case)	7	8/11 used history
2. How would you undo the previous change, i.e. make Rec_1 blue and Rec_2 red? (applying specialization for non-repetitive case)	9	8/11 used history
3. How would you resize all three rectangles to have a width of 250 and a height of 70? (applying generalization for repetitive case)	Similar to 6	10/11 used history
4. What would happen if you move the Color operation up in the history panel by 1 step? (understanding generalization)	6	11/11 correct
5. What would happen if you move the Color operation up in the history panel by 2 steps? (understanding generalization)	6	11/11 correct
6. What would happen if you move the Color operation up in the history panel by 1 step? (understanding generalization)	Variation of 6	11/11 correct
7. What would happen if you move the Color operation up in the history panel by 2 steps? (understanding generalization)	Variation of 6	11/11 correct
8. Assume the default color for a rectangle is red. What would happen if you delete the first Color operation (blue)? (understanding history)	8	10/11 correct
9. Assuming the default color for a rectangle is red. What would happen if you delete the second Color operation (green)? (understanding history)	8	9/11 correct
10. Which operations would be deleted if you delete the New operation? (understanding cascading delete)	9	11/11 correct

Table 1. Evaluation questions and results



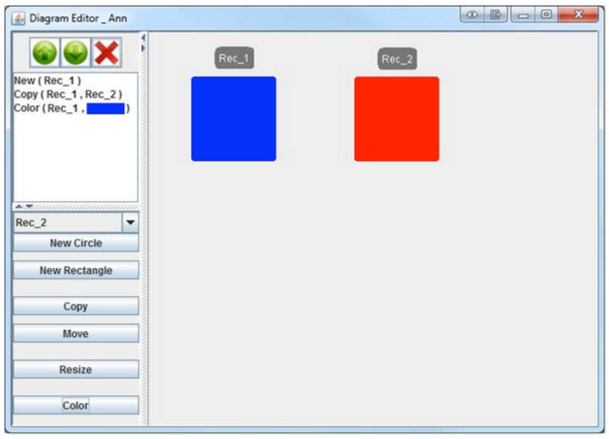
New (Rec_1)
Copy (Rec_1 , Rec_2)
Copy (Rec_1 , Rec_3)
Color (Rec_1 , 🟦)

Figure 6. Screenshot for questions about generalization



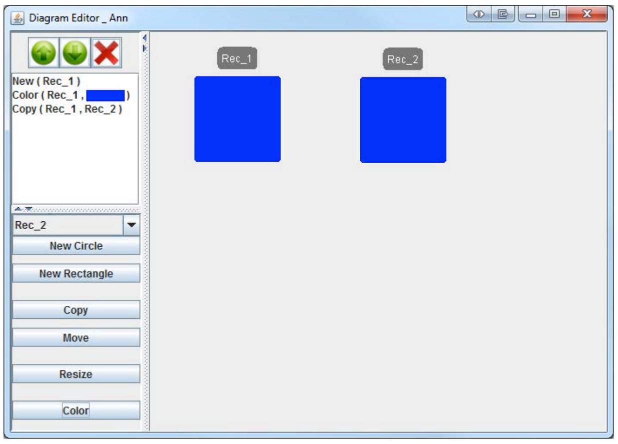
New (Rec_1)
Color (Rec_1 , 🟦)
Copy (Rec_1 , Rec_2)
Color (Rec_1 , 🟩)

Figure 8. Screenshot for questions 8 and 9



New (Rec_1)
Copy (Rec_1 , Rec_2)
Color (Rec_1 , 🟦)

Figure 7. Screenshot for questions 1, 2 and 10



New (Rec_1)
Color (Rec_1 , 🟦)
Copy (Rec_1 , Rec_2)

Figure 9. Screenshot for questions 2 and 10

Questions	Figure	History
1	7	New(Rec_1), Copy(Rec_1, Rec_2), Color(Rec_1, blue)
2, 10	9	New(Rec_1), Color(Rec_1, blue), Copy(Rec_1, Rec_2)
3-5	6	New(Rec_1), Copy(Rec_1, Rec_2), Copy(Rec_1, Rec_3), Color(Rec_1, blue)
6, 7	Variation of 6	New(Rec_1), Copy(Rec_1, Rec_2), Copy(Rec_2, Rec_3), Color(Rec_1, blue)
8, 9	8	New(Rec_1), Color(Rec_1, blue), Copy(Rec_1, Rec_2), Color(Rec_1, green)

Table 2. Histories given in the evaluation questions

There were 11 participants, who were primarily undergraduate software engineers in their fourth year of study. For each participant, the evaluation started with a short tutorial, followed by a questionnaire. The tutorial familiarized the participants with the drawing application by getting the participants to perform various step-by-step tasks, which involved generalizing, specializing and cascading delete history operations. This tutorial was necessary since the concepts we were trying to evaluate are novel, and it is highly unlikely that users would have come across them before.

The questionnaire began with 10 open-ended questions, which are given in Table 1. Each question refers to a screenshot that was given to the participants on paper. For space reasons, not all the screenshots are shown here, but it is indicated in the table which figure in this paper is similar to the screenshot shown in the questionnaire. The histories given for the questions, which are too small in the figures to read, are listed in Table 2. For the questions 6 and 7, the screenshot used in the study varies from Figure 6 in the way the rectangles are created by copying. In Figure 6, both red rectangles are copied from the blue rectangle, while in the variation, the last rectangle is copied from the first copy. For the questions 2 and 10, the example from Figure 7 is used, but in a later stage, namely after applying the change in Question 1. This means both rectangles have the same color.

The open-ended questions fell into two main categories. The first category (3 questions) had the purpose of finding out if participants would prefer to use history editing in situations where the task could be completed by either using the history rewriting method or the traditional way of using just artifact operations. For example, when using the history rewriting method, a correct answer for question 3 would be: "By resizing any of the rectangles and moving the resize operation above all copy operations." Using the traditional method, a correct answer for question 3 would be: "By resizing the rectangles Rec_1, Rec_2 and Rec_3 accordingly."

The second category (7 questions) assessed how well each participant understood the concept of history operations, i.e. whether or not they could describe what happened to a diagram when operations were moved or deleted. For example, for question 4 the correct answer is "Rec_3 would become blue", and for question 5 the correct answer is "Rec_2 and Rec_3 would become blue".

We tried to keep the tasks in the tutorial and questionnaire straightforward. The tasks are similar to common tasks that users are faced with while using a diagram editor. For example, creating multiple copies of a shape, and then editing all shapes in the same way is a scenario where history operations can be beneficial. We aimed to keep questions simple and concise with no ambiguity. Diagrams used in conjunction with the questions were simple with only relevant shapes shown.

Before running this evaluation we first tried to determine what the likely outcomes to questions would be. We expected that all testers would understand how to use history editing and would be able to answer almost all of the questions in the first category. We also realized that while the users might understand history operations and their benefits, they may still prefer more traditional ways of carrying out tasks.

The open-ended questions were followed by two Likert-scale questions. The following statements were rated on a 5-point standard scale ranging from "strongly disagree" to "strongly agree":

1. I find editing the history of operations a useful feature.
2. I would use this feature if it was included in a drawing application.

The Likert-scale questions were followed by four more open-ended questions:

1. In which situations could you imagine using this feature?
2. What did you like / find most useful about this feature?
3. What did you not like about this feature?
4. What recommendations would you give to improve this feature?

Since the evaluation contained open-ended questions and questions about the user's preferences, we did not measure the time taken by each user to complete the evaluation.

Results

According to the results in Table 1, all questions were answered correctly by a large majority of the participants. To analyze how likely users in the sampled population are to use history rewriting or answer questions about it correctly, we calculated the 95% binomial proportion central confidence interval. For the questions 1 and 2, the confidence interval is [0.43, 0.90], which means that it is statistically not clear whether a majority of the sampled population would apply history operations in these cases. This does not come as a surprise, since for these questions history rewriting did not reduce the amount of work as compared to using artifact operations. For Question 3, where history rewriting was more work efficient, the confidence interval is [0.62, 0.98]. This means that with 95% confidence a majority would use history rewriting in this case.

Questions 4-7 were designed to investigate whether users understand the concept of generalization in simple cases. The confidence interval for the proportion of the sampled population who can answer these questions correctly is [0.74, 1]. Hence, we can be sure that a majority understands generalization in such cases.

Questions 8 and 9 investigate another aspect of history, namely the effect of the delete operation. The confidence intervals are [.62, 0.98] for Question 8 and [.52, 0.94] for Question 9, meaning that again a majority understands this concept, although the result is barely significant for Question 9. Finally, Question 10 addresses a more advanced concept, namely cascading delete. This concept is clearly understood by a majority, with a confidence interval of [0.74, 1].

The two Likert-scale questions aimed at finding out how useful the participants found history operations and if they would use them if they were available in their usual diagram application. For the first question, which evaluated usefulness, 91% of the participants indicated that they thought the concepts were either useful or very useful. Similarly, for the second question, 91% of the participants said they would be likely or very likely to use these features if they were available in their usual drawing application. For both questions the 99% binomial proportion central confidence interval for a positive answer is upwards of 0.5. This means that with 99% confidence a majority of the sampled population considers history rewriting useful and would use it.

Discussion

The 11 participants did mostly have a software engineering background, or at least very good computing skills. This and the small sample size are clear limitations of the study. Future studies should include participants with a wider range of computing skills to see how they cope with the ideas of history rewriting.

The category 1 questions asked in the evaluation did not make any suggestions to the participant on how each task should be completed, save for the fact that the participants completed the tutorial immediately before the questionnaire. This meant it was up to the participants' own judgment which method they would use to accomplish the tasks. We discovered that for simple 1-step tasks, i.e. changing the color of one rectangle, participants were only slightly more inclined to use generalization and specialization compared to traditional methods. However, for more repetitive tasks that required the user to apply the same operation to many objects, i.e. coloring three rectangles blue, a greater number of participants preferred history operations over the alternative of performing artifact operations repetitively.

For the questions 4-7, all participants provided the correct answers, indicating that they understood the basic idea of generalization. However, for the more complex example in the questions 8 and 9, some participants answered incorrectly. The short tutorial carried out by the participants before the questionnaire did not include such more complex examples. This raises the question how well history rewriting is understood for longer and more complex histories, as they are likely to occur in real applications. This questions needs to be addressed in future studies.

The last 4 open-ended questions stimulated an unexpected creativity in the participants. The prototype inspired all participants to answer them in great detail, and the amount of time and effort they put into the answers surprised us. This indicates that the ideas we presented in the study were of interest and value to the participants. The participants suggested several of the extensions that we were currently working on or that are planned as future work, although we had not mentioned them. This gave us confidence that the project was heading in the right direction. For example, many participants recognized that besides ordering the artifact operations by time, other views of the history would be useful, such as grouping operations by shape. Many participants suggested visualizing the history using some kind of hierarchical structure such as a tree. This is one of the major extensions of the history view that is currently planned. The prototype used in the evaluation did not support jumping of commutative operations. However, some participants anticipated this feature by mentioning that it would be useful if a swap history operation always resulted in a visible change of the artifact.

There were many suggestions that we did not yet think about, and future research may look into them. Examples of such suggestions are: dragging and dropping operations in the history panel to reorder them, the ability to edit parameters of an existing operation, and the ability to create macros (sets of operations which could be applied to objects).

FUTURE WORK

One thing which became apparent from the usability study was that a simple textual operation history, like the one which is currently employed, is cumbersome for histories of significant length. Therefore plans are in place to research and revise the current visualization. Some options which have already been discussed include scene graphs, collapsible sections of history, and multiple history visualizations such as grouping operations by object. As a first step, the prototype has recently been extended to include a visualization of commutative and dependent operations. Selecting an operation in the history panel causes all dependent operations to be highlighted.

Furthermore, there is a large number of small or rather technical improvements. For example, copying should be supported for groups of objects, so that complete artifacts such as a blank letterhead can be copied as a whole. Symbolic layers could help to group objects and structure an artifact on a level above that of individual objects.

We will also introduce a second cursor in the history view, called display cursor. It allows users to mark a point in the history, and the artifact pane will display the state of the artifact at that point in time. The display cursor could also be used to insert new operations at an earlier point in time directly. With this feature, we can even reduce the delete operation to the swap operation in the following way: deletion means swapping an operation to the future, beyond the current display cursor. This might, however, be more of theoretical than practical interest.

CONCLUSION

The rewriting history approach, in particular the reordering of operations, satisfies important use cases and enables exciting new ways to share and combine creative work. The natural correspondence between user actions and the recorded history may help users to understand the approach intuitively, and make use of its many possibilities.

We single out three key findings of our analysis. First of all, we found that only two history operations are required for history rewriting: swapping and deleting. Secondly, swapping alone can solve all the three use cases of generalizing, specializing and merging. This particular finding has a theoretical as well as a practical consequence. Theoretically it gives us a precise way to describe the semantics of an operation such as a merge. Practically it allows us to build more straightforward tools that support powerful history rewriting features. Thirdly, the rewriting history approach cannot be reduced to common version control approaches, but can be usefully complemented by them.

Preliminary results from a user study indicate that users are excited about the idea of history operations. Users were able to understand and leverage history operations to their advantage after only a short tutorial. This indicates a gentle learning curve, and could mean that history operations are an intuitive concept.

REFERENCES

1. Agustina, A., Liu, F., Xia, S., Shen, H. and Sun, C. CoMaya: incorporating advanced collaboration capabilities into 3d digital media design tools. Proceedings of the ACM Conference on Computer Supported Cooperative Work, 2008.
2. Apiwattanapong, T., Orso, A., and Harrold, M. J. JDiff: a differencing technique and tool for object-oriented programs. *Automated Software Eng*. 14(1), March 2007.
3. Conradi, R. and Westfechtel, B. Version models for software configuration management. *ACM Comput. Surv*. 30, 2 (1998), pp. 232-282.
4. Ellis, C. A. and Gibbs, S. J. Concurrency control in groupware systems. Proceedings of the SIGMOD International Conference on Management of Data, 1989.
5. Grossman, T., Matejka, J. and Fitzmaurice, G. Chronicle: capture, exploration, and playback of document workflow histories. Proceedings of ACM UIST, 2010.
6. Hashimoto, M. and Mori, A. Diff/TS: a tool for fine-grained structural change analysis. Proceedings of the 15th Working Conference on Reverse Engineering, pp. 279-288, 2008.
7. Heer, J., Mackinlay, J., Stolte, C. and Agrawala, M. Graphical histories for visualization: supporting analysis, communication, and evaluation. *IEEE Transactions on Visualization and Computer Graphics*, pp. 1189-1196, November/December 2008.
8. Koegel, M., Herrmannsdoerfer, M., Li, Y., Helming, J. and David, J. Comparing state- and operation-based change tracking on models. Proceedings of the 14th Enterprise Distributed Object Computing Conference, IEEE, 2010.
9. Kurlander, D. and Feiner, S. Editable graphical histories. Proceedings of the IEEE Workshop on Visual Languages, 1988.
10. Mehra, A., Grundy, J. and Hosking, J. A generic approach to supporting diagram differencing and merging for collaborative design. Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ACM, 2005.
11. Mens, T. A State-of-the-art survey on software merging. IEEE Transactions on Software Engineering, pp. 449-462, May 2002.
12. Nakamura, T. and Igarashi, T. An application-independent system for visualizing user operation history. Proceedings of ACM UIST, 2008.
13. Scott Chacon. Rewriting history. In: Pro Git. Apress, August 2009.
14. Zündorf, A., Wadsack, J.P. and Rockel, I. Merging graph-like object structures. Proceedings of the 10th Workshop on Software Configuration Management, 2001.