Reflections on Undo

Roberta Mancini* Alan Dix† Stefano Levialdi*

Abstract

Undo is seen as essential element of interactive systems. However, despite its prevalence users are often confused by its behaviour and developers often apply undo inconsistently within their systems. This report focuses on single-user linear undo/redo systems It presents an abstract formal framework for modelling undo and related system extensions, a classification and taxonomy of undo and redo, and formal specifications of each class of undo/redo. This combination of formal and informal analyses clarifies the different kinds of undo/redo that are to be found in interactive systems and helps shed light on the fundamental reflexive nature of undo.

URL for related work: `http://www.soc.staffs/ac/uk/~cmtajd/topics/undo/`

1 Introduction and background

1.1 Undo

The issue of undo in user interfaces has been studied by several authors over many years (e.g. [2, 13, 21, 19, 15]). This has included both work aimed at understanding the problem, and work on implementation structures. Despite this, experiments have shown that experienced users of Microsoft Word, which has a relatively simple and easy to use undo function, still have great difficulty in working out what undo will do in some contexts [16]. Is this because we still do not have a clear idea of what undo should do, or is it simply that undo is intrinsically complex?

This is not simply a matter of theoretical interest. At the time of the earlier formulations of undo, the users of most interactive systems were either expert, or at least computer literate. Even if the users of a system with complex undo mechanisms, such as Emacs [18], did not fully understand its semantics, at least they would not be too intimidated by its often erratic behaviour. Now sophisticated multi-step undo is available on standard office systems such as Microsoft Word 6, and indeed the ability to undo with ease (not necessarily with an undo command) is seen as one of the key features of the direct manipulation paradigm [17].

In this report, we seek to clarify some of these issues by:

- building a structured taxonomy of possible linear undo/redo mechanisms;

*Dipartimento di Scienze dell'Informazione, Università degli Studi di Roma "La Sapienza", Via Salaria 113, 00198, Rome, Italy.

†Formerly at School of Computing and Mathematics, University of Huddersfield. Now at School of Computing, Staffordshire University, PO Box 334, Beaconside, Stafford ST18 0DG, UK.

- formalising a fully abstract model of undo, that is, a model of classes rather than specific undo mechanisms;
- clarifying the reflexive nature of undo and redo and how these relate to different notions of 'state'.

We deliberately do *not* demonstrate any new mechanisms or implementation methods; our aim is to understand what undo should do.

In recent years there has been substantial interest in non-linear or selective undo [3]; that is the ability to undo arbitrary commands in the interaction history. This is particularly important in multi-user undo where other users may have performed actions on a shared object between when a user performs and incorrect action and when he decides to undo it [1, 4, 6]. This report does not deal with these cases as there are enough complex issues to consider in the 'simple' case of linear single-user undo!

1.2 The nature of undo

So, what is undo? Abowd and Dix [1] make a strong distinction between the meaning of undo from the user's and system's point of view. From the system's point of view, undo is a function, and as one, does something; from the user's point of view, undo is an intention, the intention to recover a past situation. This recovery can be done employing any system function, not only undo. Nevertheless, it is often useful for the user to have a suitable tool helping him in pursuing his aim. This tool is the undo function; whether accessed via a button, function key or a menu option.

So undo is a user intention facilitated by a system function. What then does the undo *function* do? In most systems undo allows the user to reach the immediately previous state, so one could regard undo is a function which allows the user to delete, or remove the effect of the previous action. However, what happens if the previous action is an undo? Some systems do not allow you to perform the undo of the undo, some others allow it. In the latter case there are further possibilities: the undo of the undo may be used as a backtracking tool successively moving the user further into the past history of his interaction; alternatively, successive undos may make the system oscillate between two states. Moreover, other systems allow to reach not only the immediately previous state, but also choose any one in the past history.

A weak definition, which includes all the above possibilities, may be to say undo is a system function that allows the user to reach some state in the past history. This means that undo may be considered as a special case of reachability [9]. The reachability property is said to hold for a system if, starting from any system state, it is possible for the user to reach any other state by using appropriate commands. If reachability holds for a system, the user can reach *any* state, both ones that have previously been reached (already present in the action history) or ones not yet reached (possible future actions). So, reachability allows the user to move in both the directions of the action history, past and future. Undo is a special case of reachability, in the sense that it allows the user to move only in one direction of the action history, the past.

Among all the functions that a user can perform while interacting with a computer, undo, as we will explain in the following sections, is one of the most complex and its behaviour is different from any other system function. In particular, after performing an undo, the user

may find the state the system reaches is different from what he would have predicted. This inconsistency arises from several causes, some are to do with the precise function of undo. As is already evident, there are several such interpretations, and in the rest of this report we shall try to clarify these different interpretations. In addition, there are often problems due to the way undo treats the granularity of data or user actions. This inconsistency arises because undo shows something of the internal functioning of the system, of which the user is otherwise unaware.

From the above considerations we can see that the world around undo is very confused. Given the range of interpretations of undo in different applications, it is clear that there is no common understanding or definition of undo. If applications differ between one another, it is no wonder users become confused.

1.3 Models of undo

Different formal models have been proposed in the literature in order to describe undo in interactive systems [2, 13, 19, 20]. One of the most influential is the 'script' model of Archer, Conway and Schneider [2], which we will refer to as the ACS model. This is based on three streams of actions: the User History, the Active Script and the Pending Script. The User History is simply a list storing all the user's actions. Sequences of commands produce scripts; there is an immediate mapping between each script and a state as the object visualised on the screen. The Active Script is the list of those user commands which are taken into account in the current state. The Pending Script is the list of commands deleted in the Active Script by undo which are 'available' for redo (where there is a redo command).

This model can be used to discuss most (but not all) undo systems. As an example, let us consider text editor with both undo and redo. We can trace the various scripts as the user enters the actions: 'type(hi), type(everybody), undo, redo'. We will start off from an empty state. Then as the user types words they become part of the User History and also the current Active Script:

User History type(hi)
Active Script <*type(hi)*>
State hi
Pending Script < >

User History type(hi) type(everybody)
Active Script <*type(hi), type(everybody)*>
State hi everybody
Pending Script < >

When the user performs the undo action, it become part of the User History, but the previous command is removed from the Active Script (it is undone) and the state reflects this:

User History type(hi) type(everybody) undo
Active Script <*type(hi)*>
State hi
Pending Script <type(everybody)>

Notice how although the command 'type(everybody)' has been removed from the Active Script it is still recorded in the Pending Script. This is so that we know what a successive redo action should do:

> **User History** type(hi) type(everybody) undo redo
> **Active Script** *<type(hi), type(everybody)>*
> **State** hi everybody
> **Pending Script** < >

This model emphasises several other points which we will address in progressive detail through the report:

- We must consider two kinds of history, one recording all user actions (which is always added to by user actions), the other recording the actual 'script' of commands which are deemed to have happened to give rise to the *observed* state of the system.
- We must also consider two kinds of commands, normal commands (such as 'type(hi)') and extra ones to allow undoing (such 'undo' and 'redo').
- The scripts used in the model are not necessarily recorded in actual undo systems – they serve to describe the behaviour, not the implementation of such systems.
- The 'state' in the ACS model is not the entire state of the system. It corresponds to the state of the system when one ignores undo. Some extra history information must be stored.

1.4 Structure of the report

The rest of this paper we consider the interpretation and modelling of undo and redo. In the next section we will look at the issue of the two types of commands and two types of state in greater detail, and also at the reflexive nature of undo: looking in on and acting on the history of interaction. In section 3, we will develop an abstract formal framework which can be used for the modelling of undo and redo. Section 4 presents a taxonomy of undo followed by a formal model of each class of undo system in section 5. Sections 6 and 7 follow the same pattern presenting a taxonomy and then formal models of redo. Finally in section 8, we consider critical issues and reformulate our informal notions of redo.

2 Undo and the original system

Notice that some parts of the ACS model refer to the original system without undo: the active script and the state. Adding undo to such a system in some way enhances or extends it, but we expect the original system still to be 'in there' somewhere. It is central to the understanding of undo that one draws a distinction between this underlying system and the enhanced system with undo. The reason for the importance of this distinction is the reflexive nature of undo. First of all, let's look at some of the problems that arise if this is ignored.

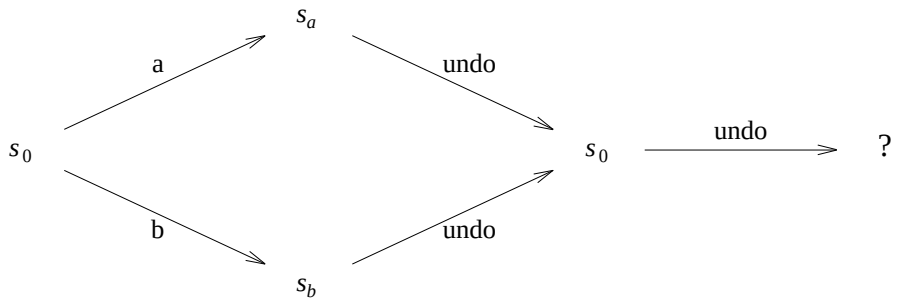

Figure 1: Undo of undo?

2.1 The reflexive nature of undo

An obvious definition of undo is to say that following any command by undo makes it as if the original command had never happened. Or, in other words, the state of the system after the undo is the same as the state of the system before the command. We can write this formally as:

$$c \frown undo \sim null \qquad (\textit{strong-cu})$$

This uses the strong equivalence ($\sim$) from [9], which says that in all contexts the command c followed by $undo$ has the same effect on the state as the null command (that is, doing nothing). In other words, if we see this pair in any command history we may safely remove it.

This definition of undo looks nice algebraically and appears to correspond to one's intuition. To be truly general, one would like this to hold for all commands c, even including undo itself. However, because of the special nature of undo, it is often the case that properties that hold for other commands do not hold for undo and vice versa. Hence we use *strong-cu* to refer to the property for all non-undo commands, and introduce a similar definition for the case when undo acts on itself:

$$undo \frown undo \sim null \qquad (\textit{strong-uu})$$

This *strong-uu* property captures the case when undo is truly reflexive and acts equally on itself as well as on non-undo commands.

The combination of the two (strong-cu and strong-uu) gives an undo property which has been called *thoroughness* [21]. However, it turns out to be effectively inconsistent. Yang proves that the two common forms of undo system do not satisfy this strong undo property [21]. In fact, it is shown in [8, 9, 10] that *no* undo system can satisfy this property except those where the underlying system has at most two states! The proof of this is quite straightforward, but somewhat counter-intuitive and instructive of the nature of undo.

The essence of the proof is summarised in figure 1. The top and bottom routes round this diagram consider two different potential interactions from an arbitrary state of the system s_0. Call the state obtained if a were executed s_a, and the one if b were executed instead s_b. Now consider the effect of undo on either state. Both must go back to the original state s_0, as both $a \frown undo$ and $b \frown undo$ are equivalent to the null command (doing nothing). Finally, consider what happens if undo is issued from the state s_0. Arguing from the top interaction history, we know that $undo \frown undo$ should have no effect, so the resulting state is s_a. From the lower interaction path, one would conclude that executing undo in the state s_0 should lead to s_b. Which is right?

Well, if the strong undo property really holds of the system, both must be right. That is, $s_a = s_b$. But as a and b were arbitrary commands, that means the effect of any command in the state s_0 is the same. Arguing a little further and noting that if a and b are arbitrary, one of them could even be undo, we see that the system can have at most two states, with all commands (and undo) simply toggling between them. That is, the strong undo property is impossible to satisfy for any realistic system.

In other words, although undo is reflexive in the sense that it looks in on the interaction history of the system, it cannot be entirely reflexive, treating itself on a par with other commands.

2.2 Two kinds of state

The first reaction of many people on seeing the above proof is "well, it looks OK, but I use a system that satisfies the property". They think there is some mathematical sleight of hand at work, but that it isn't really as bad as the formal proof seems to suggest. UNIX users often cite vi as a counter-example and PC and Mac users cite Word 5. In each case, the undo command toggles back and forth between two states of the system. (This form of undo is sometimes called flip-undo [19] and we will return to its properties later in this paper.) If undo is followed by another undo, the system appears to be in the same state as before the first undo. So, why the discrepancy between the formal proof and practical experience? The answer lies in the use of the word 'state', when we say that the state of the system is the same after the command pair $c \frown undo$ or the command pair $undo \frown undo$.

One definition of 'state' is the one we get in the 'state' component of the ACS model. This corresponds to the state of the system if there were no undo. We'll call the set of such states S.

However, in order to be able to perform undo, the system must store additional information, often some sort of history or record of past states. That is, the full state of the system contains more than in S. This complete state of *all* the system, including the bits needed for undo, we will refer to as S^a.

What the proof shows is that you cannot satisfy the strong undo property with respect to the full system state S^a. In the examples of vi and Word 5, the undo systems do, in fact, satisfy the strong-uu part of the property (except for minor differences in the display), but do not satisfy the strong-cu property. Although the S part of the state is the same after an undo, the full state is different.

2.3 Two kinds of command

In a similar fashion we need to divide all the user's actions into two classes: the first is made by all the functions that are strictly related to the user's task; the second is made by any functions that allow the user to modify the past interaction. Yang refers to the latter as the recovery commands [21]. We have seen already with the *strong-cu* and *strong-uu* properties that it is useful to consider these classes separately.

Indicating by A the set of user actions, we have that $A = (C \cup R)^*$, where C is the set of allowable commands and R is the set of recovery commands, including any different kinds of undo and possibly some sort of redo function.

In the case of a single undo command, this simplifies to $A = (C \cup \{undo\})^*$.

We will use H^a to denote the set of sequences (or histories) of actions ($H^a = A^*$), and H for the set of simple command histories ($H = C^*$). That is, H^a corresponds to the 'user history' part of the ACS model, and H corresponds to the commands in the 'active script' or, in other words, the commands issued to the system as if there were no undo.

The subdivision of user actions into commands and undo is naturally generated by the user's different aims:

command The user's aim is to modify an object.

undo The user's aim is to delete a modification, the effect of a command on an object: in other words, to modify interaction itself.

Starting from the above distinction, it is possible to provide a functional description of both ordinary commands and undo.

An ordinary command $c \in C$ is a function that modifies an object of interest, that is, an object directly related to the task the user is performing. Such objects are those that are in the state of the system, even if we ignore undo; that is, S. So, the primary purpose of commands is their effect on S. This can be modelled using a $doit$ function (state update) [9]. We have that $doit(s, c) = s'$, that is, performing the command c, we can pass from the state s to s'. Now, s' is a new state, typically distinct from all the previous states, and both s and s' belong to the set of states S. In the ACS model, $doit$ corresponds to the result of the natural mapping between the Active Script and the actual situation of the document, ignoring the history. Note that this $doit$ function only tells us about the effect of ordinary commands on the state of the system *without undo*. They will also have some effect on the rest of S^a: for example, the command may be added to the end of a history list. That is, there is a full state update function $doit^a$, which acts on S^a and determines the complete behaviour of the system. In the next section we will define formally the relationship between $doit$ and $doit^a$.

Turning to undo, the object of interest of undo depends on which definition we consider. However, its data of interest are not the same objects on which the commands act. Instead, such data may be commands or actions; that is, if we consider undo without redo, the application domain may be the command history (H) or the action history (H^a).

In the first case, undo can be defined by a function $U : H \rightarrow H$ acting on previous commands to reverse their effect. The effect of undo itself cannot be reversed, since it does not belong to its domain of definition. Effectively all past undos are forgotten, except for their effect in having reversed previous actions. Such a system can have no redo function, and undo acts as a pure backtracking tool.

Alternatively, the domain of interest of undo may be the complete action history (from H^a). In this case, undo can be defined by a function like $U : H^a \to H^a$. That is, the system regards undo as a command also. However, as we saw at the beginning of this section, the behaviour of this function on the action history cannot be uniform and, in fact, there must be a different system behaviour when performing undo after a command than after a previous undo. This is exactly what we see in all systems with undo. When the previous action has been a command, then we expect undo to act upon it by reversing its effect. In this case, we have $doit^a(s, undo) = s'$, where $s, s' \in S^a$, yet s' is not exactly a previous state, but in some sense an equivalent one, because the system keeps some trace of the preceding activity. When performing undo of undo, different systems have different behaviour: some of them simply do not allow it, others consider undo of undo as the redo function. We will consider redo later, but the former case, of simple single step undo, can be described informally as:

$$U(h \frown a) = \begin{cases} h & \text{if } a \text{ is a command from } C \\ \text{not allowed} & \text{otherwise} \end{cases}$$

The peculiarity of undo is that it is not a command but a meta-command, its effect depends on the context of previous commands, and, being meta-command, its structure is quite different from that of the ordinary commands. When using undo as a command, some aspects of this reflexive structure are revealed to the user, giving rise to problems of inconsistency and even apparent randomness, especially if the user is expecting a different kind of undo behaviour. Moreover, since the domain of interest of undo is an action or command history, when using undo the user is not simply interacting, but instead interacting with interaction.

3 An abstract formal framework for undo

In the previous section we distinguished the state and commands of the original system without undo, from the full system state and action history when undo (and possibly other recovery commands) has been added. In this section we will formalise these two views of the system and, most important, discuss their relationship.

Imagine your company has developed a word processor, but it doesn't have an undo facility. You give it to your development team and ask them to add undo. Six months later they come to you with a fully functioning system with up to 10,000 levels of undo. Unfortunately, it is not a word processor but a spreadsheet. It is obvious they have done something wrong. However, what if the differences they made when adding undo were more subtle? They may have had to make major changes to the internal structure of the program in order to implement undo. How can you be sure that it is the 'same' system after undo has been added?

We will define formally a relationship between the two models: the system with undo and the system without undo. This relationship, which we call *conservative encapsulation*, captures the idea that the original system is, in some way, still there 'inside' the full system with undo.

We consider first the system without undo, then look at the full system, and finally the relationship between the two. The model we will use is a form of the PIE model [7], using multiple levels of abstraction as found in [9].

3.1 System without undo

We have already partly introduced the formal model of the original system in the last section. The set of states we call S, and the set of 'ordinary' commands C. The two are linked by a state update function $doit$

$$doit : S \times C \rightarrow S$$

and the system starts from an initial state s_0.

As in previous work we can derive from this function two other functions: $doit^*$, obtained by iterating $doit$, and I, the *interpretation* function of the PIE model:

$$doit^* : S \times H \rightarrow S$$

where

$$doit^*(s, <>) = s$$
$$doit^*(s, h \frown c) = doit(doit^*(s, h), c)$$

This iterated version tells you the effect of a whole sequence of commands. Recall that the sequence of commands, written as H, the command history, is defined by $H = C^*$, the set of finite sequences of C.

The interpretation function is simply the iterated $doit$ starting from the initial state:

$$I : H \rightarrow S$$

where

$$I(h) = doit^*(s_0, h)$$

We will also use a dot to represent the 'curried' version of a $doit$ function:[1]

$$doit(., c) : S \rightarrow S$$

where

$$doit(., c) = \lambda s \bullet doit(s, c)$$

3.2 System with undo

When we consider the system with undo, as we noted, the state space increases. The set of full states we call S_a and the set of actions $A = C \cup R$. There is a corresponding state update function $doit^a$ and initial state s_0^a. As with the original system we can define an iterated version $doit^{a*}$ and an interpretation function I^a.

It is important to note that this full state will extend the original state, not in the sense that there are extra possible states (i.e. *not* $S \subset S^a$), but in the sense that each state of the full system has some component (or effectively such) that corresponds to a state of the original system. That is, there is some projection function $proj$, which, given a state of the full system, gives a corresponding state of the original system.

[1] Currying is a technique used in functional programming and lambda calculus to simplify the presentation of complex formulae. Some of the parameters of a function are fixed, giving a function with fewer parameters. In this case, we are fixing the command parameter of $doit$, yielding a function $doit(., c)$, which only has one parameter, a state.

$$proj : S^a \rightarrow S$$

Typically, the full state contains some form of history information. For example, a particular undo system might store the 'normal' state and also the command history (active script). That is, its state would be given by:

$$S^a = S \times H \qquad \textit{(example state)}$$

The projection function would then be:

$$\forall < s, h > \in S^a \bullet proj(< s, h >) = s \qquad \textit{(example projection)}$$

The exact way in which the original state is extended, and the nature of the projection function, will differ between undo functions.

In section 2.1, we used an equivalence relationship '$\sim$' to define the *strong-cu* and *strong-uu* properties. This was defined loosely at the time, but has a precise definition in terms of $doit^a$. Given any two histories h and h' from H^a we say that $h \sim h'$ if:

$$\forall s \in S^a \bullet doit^{a*}(s, h) = doit^{a*}(s, h')$$

So, for example, *strong-uu* can be restated in terms of $doit^a$:

$$\forall s \in S^a \bullet doit^a(doit^a(s, undo), undo) = s \qquad \textit{(strong-uu)}$$

The equivalence '$\sim$' often gives more compact and more algebraic formulations of properties, but is identical to the above functional formulation.

3.3 Encapsulation

Not only must the extended system have an undo command, but it must in some sense preserve the original system inside. We capture this in two stages: first of all the idea of *encapsulation* and then that of *conservativeness*.

We have already related the states with a projection function $proj$. For an encapsulation we also require a mapping between the input histories, which from any action history of the full system gives an effective command history. We call this function eff. It corresponds to the mapping in the ACS model that determines the active script from the user history.

Formally, we say that the augmented system $< H^a, S^a, doit^a, s_0^a >$ is an *encapsulation* of the original system $< H, S, doit, s_0 >$ if there exist two functions $proj$ and eff such that:

(i) $proj : S^a \rightarrow S$

(ii) $eff : H^a \rightarrow S$

(iii) $\forall h \in H^a \bullet proj(I^a(h)) = I(eff(h))$

The last condition says that the part of the state corresponding to the original system is just as if you had executed the effective history. Indeed, the system may actually be implemented by using the original update functions on this part of the system state. Note that, this condition says nothing about the way in which the effective history is related to the action history, merely that it and the projected part of the state 'agree'.

The conditions for an encapsulation can be summarised by the commuting diagram in figure 2. The two sides of the above equation correspond to the two paths round the diagram.

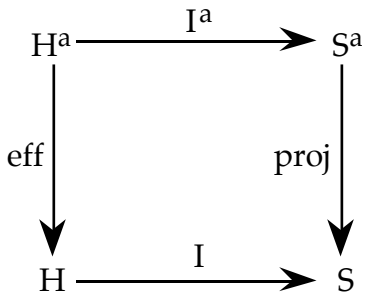

Figure 2: Encapsulation.

3.4 Conservativeness – the effect of ordinary commands

The encapsulation condition says that the original system is still in there. However, we have so far set no conditions other than that the effective history and the projection in some sense agree. We want to say more. Obviously the new commands may have arbitrary behaviour, but we expect the original commands to behave as they always did on the original part of the state. In keeping with other areas of formal specification, we regard this as a *conservativeness* property – the original system is conserved within the extended system.

Looking first at the state, we expect that: (i) the initial state of the full system (s_0^a) corresponds (via the projection function) with the initial state of the original system (s_0); and (ii) the effect of applying a command to the full state parallels that of applying it to the projected form of the state. Formally:

(i) $proj(s_0^a) = s_0$

(ii) $\forall c \in C, s \in S^a \bullet proj(doit^a(s, c)) = doit(proj(s), c)$

Again, this can be captured in a commuting diagram, figure 3. The main part of the diagram corresponds to condition (ii), and the small triangle on the left to condition (i). The '1' refers to the set of one element and the arrows labelled 's_0^a' and 's_0' are constant mappings (from the single element of '1'). This is simply a formal trick that allows us to include this information on the diagram. Also note that the functions on the top and bottom of the diagram are the curried versions of the appropriate $doit$ functions. They are for a particular command c, and strictly one can imagine a copy of this diagram corresponding to every such command.

In a similar fashion we expect the effective history to behave in a sensible fashion where ordinary commands are concerned:

(i) $eff(<>) = <>$

(ii) $\forall c \in C, h \in H^a \bullet eff(h \frown c) = eff(h) \frown c$

That is, (i) the effective history corresponding to an empty action history should be empty, and (ii) adding an ordinary command to the action history adds the same command to the effective history. These conditions are captured in the commuting diagram, figure 4.

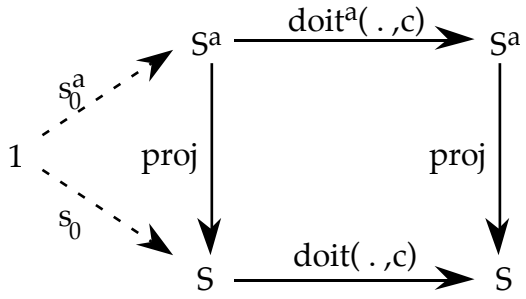

Figure 3: Conservativeness of state.

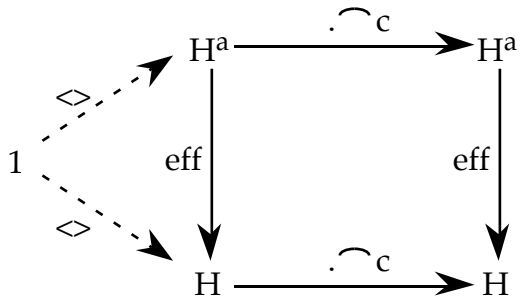

Figure 4: Conservativeness of effective history.

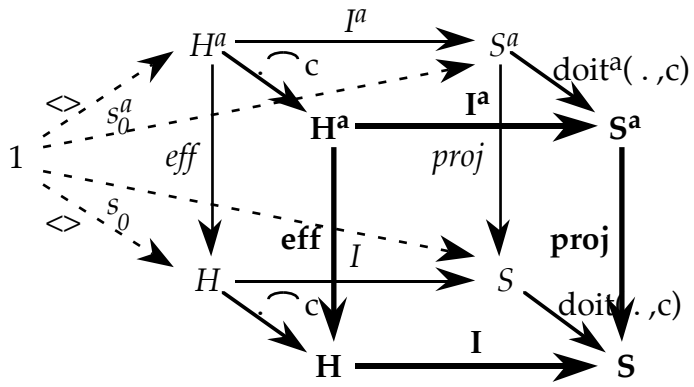

Figure 5: The cube.

3.5 Conservative encapsulation – the cube

If all three diagrams commute, we will say that the augmented system:

$$< H^a, S^a, doit^a, s_0^a >$$

is a *conservative encapsulation* of the original system

$$< H, S, doit, s_0 >$$

with respect to the two functions $proj$ and eff. The whole set of conditions can be captured in a single commuting diagram, figure 5, which we call 'the cube'. This diagram is rather complicated to read on its own, as it includes all the rest. The front and back are two copies of figure 2. The left is figure 4 and the right figure 3. To make it easier to read, the legends at the back are italicised and those at the front emboldened.

The cube has six faces: four correspond to the commuting diagrams, but that leaves the top and bottom. Drawing the bottom on its own gives the diagram in figure 6. This refers only to the model of the original system, and upon examination is simply a restatement of the construction of I from $doit$. The top triangle is the initial condition that

$$I(<>) = s_0$$

and the square corresponds to the iterated case

$$I(h \frown c) = doit(I(h), c)$$

The top is similar, except that it refers to the full system.

So, both the top and the bottom of the cube commute by the definitions of I and I^a. This is important, as it suggests that some of the faces of the cube are redundant, in the sense that they are implied by the others. This is indeed the case, and the right side of the cube, corresponding to the state conservativeness diagram (figure 3), can be derived from the other faces. That is, if we know that the encapsulation diagram (figure 2) and the history conservativeness diagram (figure 4) both commute, then we can prove that the state conservativeness diagram (figure 3) must also commute. This proof is given in appendix A.

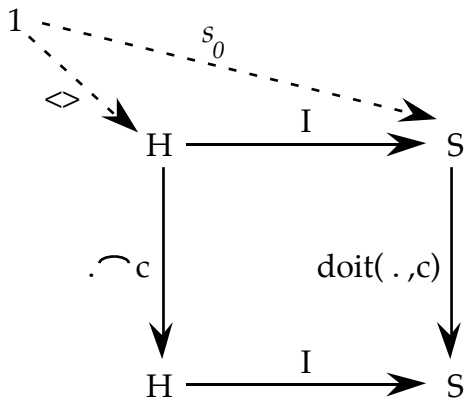

Figure 6: Bottom of the cube.

So, to verify that a particular undo system is indeed a conservative encapsulation, it is sufficient to show that it satisfies the encapsulation conditions and that the effective history behaves appropriately.

3.6 Summary of formalisation

We have formalised the difference between the state of the 'original' system before undo and that of the full system with undo. Furthermore, we have made precise the relationship between them as *conservative encapsulation*. This embodies the idea that the semantics and properties of the original system are preserved within the full system. That is, we can avoid both the extreme case of a spreadsheet with undo being delivered instead of the word processor, but also more subtle cases of the behaviour being altered by adding undo.

We have not, so far, considered the effects of undo commands themselves. Can we say anything about them without committing ourselves to specific classes of undo mechanisms? In fact, we can say something, but we will leave this until we have studied different kinds of undo in more detail.

4 Single-action and multiple-action undo

In section 2, we began to classify different kinds of undo based on whether undo is self-applicable, (i.e. if it belongs to its definition domain), or not. We have not explicitly considered redo (although the formal framework is valid for all types of recovery) and we will delay full consideration of redo until section 6.

We have also been intentionally vague as to the *scope* of undo, which varies markedly between particular undo systems. This scope has two main aspects. Firstly, the number of times undo can be applied: single-undo or multiple-undo, where by single-undo we mean the ability to apply undo only once, while by multiple-undo we mean the ability to apply undo successively. Secondly, the number of actions that may be undone at one step: many systems allow only a single action to be undone at a time, but other systems allow multiple actions to be undone at each undo step.

Repetition \ Granularity	Single action	Multiple action
Single undo	(i) Undo only the last command. Undo of undo is not allowed	(iii) Undo a block of actions. Undo of undo is not allowed
Multiple undo	(ii) Undo only the last command. Undo of undo as backtracking	(iv) Undo a block of actions. Undo of undo as backtracking

Figure 7: A taxonomy of undo function. The rows represent the granularity of undo, the columns represent its repetition.

The latter is a classification based on undo granularity, that is, how many actions may be undone at any step: one or many. Note, though, that this is different from the granularity issue that caused problems for users in the study by Wright et al. [16]. In that case the issue was how many raw user keystrokes and button presses are regarded as a single action by the system; that is, what the system has 'decided' is the minimum granularity for undo. In the case of multiple-action undo, the user 'decides' how many actions to undo at one step. In other words, at a low level the system determines the granularity of undoable actions, whereas the user determines the granularity in terms of the number of such actions to be undone. Unfortunately, because of the way undo 'digs' beneath the surface of the system implementation, several different sorts of granularity are important and we have to ignore some in order to understand others.

Figure 7 summarises the classification based on the above two distinctions: single/multiple-undo and single/multiple-action. In it we identify four classes of systems:

(i) *single-undo/single-action*, for which it is possible to apply undo only on the last performed action and undo of undo is not allowed;

(ii) *multiple-undo/single-action*, for which it is possible to apply undo on the last entered command and undo of undo is used as a backtracking tool;

(iii) *single-undo/multiple-action*, for which it is possible to apply undo only on the last block of n actions and undo of undo is not allowed;

(iv) *multiple-undo/multiple-action*, for which it is possible to apply undo on a block of n commands and undo of undo is used as a backtracking tool.

It is easy to find examples of systems in three of these classes: for instance, the standard single step undo (i) is found in many spreadsheets, graphics packages and word

processors; (ii) describes the behaviour of the 'back' command in HyperCard; and multiple-action/multiple-undo (iv) is supported via a pull down menu in Word 6. However, systems of type (iii), the top right of the diagram, seem to be absent. Why is this? In principle, it would be possible to produce such a system, but it would be silly to do so. We will see why in a moment.

One factor that differs between these kinds of undo is the amount of information they have to store in order to be able to do an undo. In the case of (i), single-undo/single-action, only the current state and previous state need to be held. Every command *commits* the previous commands, in the sense that they can no longer be undone. This is a backward commitment point, as it limits the amount the system can go 'backwards' in time to previous states. In contrast, (ii) and (iv), which both allow multiple-undo, have no backward commitment points; it is always possible to go as far back as you like. Amongst other things, this means that systems of type (ii) and (iv) must both store similar amounts of history information.

Backward commitment points are bad news for the user, as they limit the possibilities for recovery. However, they are good news for the developer, as a commitment point limits the amount the system has to store and hence the cost of the undo. In the extreme, storing *everything* can be very expensive (even when implemented carefully using 'deltas'), so many systems have slightly weaker forms of (ii) or (iv) where there is a limit on the number of commands that can be undone (e.g. one hundred commands in Word 6), or on the total resources used to store history information (e.g. Emacs, which has a large byte count limit). However, we will regard these as effectively falling into the relevant category, just as we regard a spreadsheet as being able to handle arbitrarily large sheets even if there is some resource limitation.

Looking at the concept of backward commitment points, it is clear why it is unusual to find systems of type (iii). Such a system would have no backward commitment point so long as only ordinary commands were used. So, like (ii) or (iv), it would, in principle, have to remember the complete history of the interaction. However, after a single n action undo, it would no longer be possible to go back beyond those n actions. That is, the action of doing an undo would establish a backward commitment point. Such a system would have all the disadvantages of (ii) or (iv) in terms of potential cost of maintaining history information, while making things worse for the user by establishing backward commitment points, reducing the possibility of recovery. It is not surprising type (iii) systems are rare!

The relationship between (ii) and (iv) is also rather interesting. To see this let's consider two informal definitions of undo:

Definition 1 *Undo is a system function that allows the user to reach* the *previous state*

Definition 2 *Undo is a system function that allows the user to reach* any *previous state*

Definition 1 clearly covers type (i) undo (single-undo/single-action) and type (iv) clearly falls under definition 2. What about (ii)? On the one hand, a single undo always gets back to *the* previous state. However, because the user can apply undo repeatedly it is possible to get back to *any* previous state: one can always get the effect of a single n-step undo by doing n single-step undos. So, the difference could be seen as one of *task migration* [10]; that is, the same objective can be reached either by the user or the system. Furthermore, given the mapping between physical actions and logical actions is rather a matter of taste,

one could even regard the user pressing the 'undo' button n times as being equivalent to a single logical n-step undo action!

So, to some extent, (ii) and (iv) give the user equivalent power, but with a different user interface. In fact, for n-step undo the user interface issue is particularly complex. When you perform an action you want to have some idea of what the action is going to do (predictability). Similarly, when you perform an undo, you would like some idea of what will happen. For single-step undo this can be difficult, as was shown by the study of Wright et al. [16]. However, for n-step undo things are far more difficult. Even if you have a very clear idea of the granularity of each command, can you remember just how many commands back lies the state you are trying to return to? For class (ii) you can simply go back until you notice that you are in the right case, but for n-step undo it is essential that the system gives some way of determining how many steps to go back. For example, in Word 6 the previous actions are represented in the undo menu.

Having an n-step undo, whether supplied by the system (iv) or by multiple user commands (ii), makes it possible to go back too far by accident. For case (ii) you would probably notice and, at worst, undo one step too many. In case (iv) the potential damage is greater. For multiple undos, redo is not a luxury, but a necessity.

5 Formal behaviour of undo

In section 3, we considered how the meaning of ordinary commands was preserved within an undo system. Now we turn to formalising the effects of the undo command itself, based on the two general definitions of undo we had in the last section, starting with definition 1 (the more restrictive).

5.1 Definition 1 – single-action undo

This definition expresses simple one-step undo and can be formulated very easily in the model developed earlier. However, we must be careful with the interpretation of the word 'state' in the definition "... *to reach* the *previous state*". Naively, we could say that the *full* state of the system is as it was before the last command:

$$doit^{a*}(s, c \frown undo) = s \qquad \text{eqn. 5.1}$$

Which is precisely the formal definition of the *strong-cu* condition we first considered in section 2.1.

$$c \frown undo \sim null \qquad (\textit{strong-cu})$$

If this were true, then the system would have forgotten about the command c totally, and so no form of redo would be possible. In fact eqn. 5.1 exactly captures the specific case where undo can be used as a pure backtracking tool with *no redo*, and in [14] it is proved that any undo system that satisfies this condition is equivalent to this pure backtracking undo.

This fact seems obvious, but as with many such 'obvious' things, really proving it highlights many non-intuitive features of undo. In particular, one has to be very careful about the idea of 'equivalence': there is not a *single* system corresponding to backtracking undo; instead, for any original system without undo, there is an application of the backtrack undo

policy. Although the full proof is too long we will give the definition of backtrack undo policy as a generic construction at the end of this section.

As an alternative, we can formulate definition 1 using the 'normal' component of the state, rather than full state:

$$\forall s \in S, c \in C \bullet proj(doit^{a*}(s, c \frown undo)) = proj(s) \qquad \text{eqn. 5.2}$$

We could write this in a similar way to the *strong-cu* condition, by introducing an equivalence based on commands being equivalent on the projected state: *strong-cu* condition

$$c \frown undo \sim_{proj} null \qquad (\textit{weak-cu})$$

Note that this is similar to eqn. 5.1, but only says that the projection onto the original component of the state is restored. It is thus far weaker than eqn. 5.1, both intuitively and in the formal sense that if eqn. 5.1 is true then eqn. 5.2 follows. A slightly different formulation is obtained if one instead focuses on the effective history:

$$\forall h \in H^a, c \in C \bullet eff(h \frown c \frown undo)) = eff(h) \qquad \text{eqn. 5.3}$$

This seems quite an intuitive formulation; in ACS terms it is saying that the last command in active script is removed by the undo. This is again stronger than eqn. 5.2 but, counter to intuition, it is not strictly weaker than eqn. 5.1, because the full state of the system may not store information equivalent to the effective history. This is important in understanding both our generic formulation of undo and, indeed, the ACS model. The effective history is *not* part of the system, but part of the *justification* that the system is an acceptable undo. It and the projection function are talking *about* the particular undo system.

As an example, consider a text editor with a backspace key ($\leftarrow$) as well as an undo key. We might expect the effective history for '$a\, b \leftarrow undo$' to be simply 'ab', but it would be quite reasonable to also have it as '$a\, b \leftarrow b$'. That is, the undo is translated into an equivalent action. Indeed, in many undo systems this is precisely how it is *implemented*. Equation 5.3 demands that not only does undo work in a particular way, but we also explicate it appropriately.

Notice that none of these capture the strongest notion of defn. 1, that *only* and *at most* the previous state can be reached – that is, multiple undo is not allowed. The formal definitions are permissive: saying what you can do, but not restrictive: saying what you cannot do. Such restrictive conditions are hard to state over the state, but can be formulated using the effective history. We can say that the effective history is never more than one less than the longest it has ever been:

$$len(eff(h)) + 1 \geq \max_{h' < h} len(eff(h'))$$

Note that the less than or equal relation '$\leq$' is being used as a shorthand for 'is an initial subsequence of'. That is:

$$h' \leq h \Leftrightarrow \exists h'' \textbf{ st. } h' \frown h'' = h$$

5.2 Definition 2 – linear undo

Definition 2 captures the more general case where undo may recover some previous state (but not necessarily the last). This can also be expressed formally by using the effective history. Again, we have to be careful to look to the 'normal' part of the system state, captured either by the projection function or by focusing on the effective history.

This time we want to say that after an undo command u (of which there may be several different kinds), the effective history is the same as some previous one (and hence the 'normal' part of the state is the same).

$$\forall h \in H_A, u \in U \bullet \exists h' \leq h \textbf{ st. } eff(h \frown u) = eff(h') \qquad \text{eqn. 5.4}$$

Note that this exactly captures the idea of *linear* undo as opposed to selective and other non-linear undo as discussed in the introduction. Equation 5.4 is true of *any* linear undo system. It includes as simple special cases both single-action and backtracking undo, but also much more sophisticated forms of undo–redo such as that found in Word 6.

Note also that this captures not only the expected effect of undo commands, but of *all* recovery commands including redo. The incorporation of the effective history and projection functions in the encapsulation captures the fact that adding recovery commands should not allow states of the underlying system which *could not* have happened without them. Equation 5.4 says that no amount of undos or redos will get to a state that *has not* happened already in the current interaction.

5.3 Selective undo

Although it is not the focus of this paper, we can, in fact, express the case of selective undo in a very similar fashion, namely:

$$\forall h \in H_A, u \in U \bullet \exists h' \leq h \textbf{ st. } eff(h \frown u) \preceq eff(h')$$

Note that the only difference between this and eqn. 5.4 is that the equality in eqn. 5.4 has been replaced with '$\preceq$'. This represents a weak notion of partial subsequence, where $t' \preceq t$ if the sequence t' is the same as t with some elements missing (but those that are left in the same order).

This does seem to capture intuitively the idea that selective undo removes the effect of some of the previous commands, not necessarily the last ones. However, it is not exactly right, as selective undo systems typically have to modify the effect of later actions in some manner. For example, undoing 'b' from the sequence '$a\,b\,c$' would normally result in an effective history '$a\,\hat{c}$' where '$\hat{c}$' is not exactly the same as 'c'. This is because the meaning of later commands often depends on the previous commands. This issue is covered in more detail in [12].

5.4 Specific undo policies

The implementation of an undo system will typically be very dependent on the underlying application. However, the idea of a single-step undo, or an undo system with indefinite backtracking, exists independently of the particular application. We can thus formally specify these in an application-independent manner. These specifications take the form of a construction that, given a model of the underlying system, will generate a new system with

undo, and also eff and $proj$ mappings to form a conservative encapsulation (as we said any sensible undo system must be). We will omit proofs of conservativeness here.

In each case, the underlying system will be assumed to be a PIE model with commands C, history H and associated $doit$ function, initial state s_0 and interpretation function I. No assumptions about the underlying system will be made, apart from the fact that it can be described in these terms (which rules out collaborative and real-time applications where different notions of undo are required [1]).

For both the single step and the backtracking undo, the command set consists solely of the original commands plus the special undo command. We will use superscripts s and b for the elements in the models of each. So we have:

$$\begin{array}{lll} C^s & = C^b & = C \cup \{undo\} \\ H^s & = H^b & = (C \cup \{undo\})^* \end{array}$$

Single-step undo

First of all, single-step undo. It needs to remember the previous state, so the full state of the undo system will have two copies of the ordinary system state. However, at the initial state and after an undo, the 'previous' state is inaccessible (this is *single*-step undo). To make this explicit, we will use an undefined element, $\perp$. We will call the resulting state S^s with initial state s_0^s

$$\begin{array}{l} S^s = (S \cup \{\perp\}) \times S \\ s_0^s = < \perp, s_0 > \end{array}$$

The second part of the state corresponds to the 'current' state, and the first part to the past state (which may be undefined). The projection function is trivial, simply peeling off the last component:

$$\begin{array}{l} proj^s : S^s \rightarrow S \\ \text{where } proj^s(< p, s >) = s \end{array}$$

The state update function $doit^s$ acts on this state. For ordinary commands, the 'current state' part is updated and the 'previous state' component becomes the old current state. For undo, the 'current' state component becomes the old 'previous' state and the 'previous' state component becomes undefined (disallowing further undo).

$$\begin{array}{lll} doit^s : S^s \times C^s \rightarrow S^s & & \\ \text{where} \quad doit^s(< p, s >, c) & = < s, doit(s, c) > & c \neq undo \\ \qquad doit^s(< \perp, s >, undo) & = < \perp, s > & \\ \qquad doit^s(< p, s >, undo) & = < \perp, p > & p \neq \perp \end{array}$$

The effective history is defined inductively by cases: ordinary commands simply add to the effective history (the conservativeness condition) and undo removes the last command from the effective history *only if* the last command was not undo, and otherwise leaves it unchanged:

$$
\begin{array}{lll}
eff^s : H^s \to H & & \\
\text{where} \quad eff^s(<>) & = & <> \\
\quad eff^s(h \frown c) & = & eff^s(h) \frown c \qquad c \neq undo \\
\quad eff^s(undo) & = & <> \\
\quad eff^s(h \frown c \frown undo) & = & eff^s(h) \qquad c \neq undo \\
\quad eff^s(h \frown undo \frown undo) & = & eff^s(h \frown undo)
\end{array}
$$

Note how the effect of undo cannot be defined solely in terms of the current effective history (from H), but needs the full action history H^s.

Backtracking undo

The backtracking undo can be defined in a similar manner. This time all the past states need to be remembered to allow multi-step undo. To capture this the full state has a (non-empty) sequence of past states, and the projection function merely peels off the last one: the current state.

$$
\begin{array}{l}
S^b = S^+ \\
s_0^b =< s_0 > \\
proj^b : S^b \to S \\
\text{where } proj^b(hs) = last(hs)
\end{array}
$$

The state update function $doit^b$ simply applies ordinary commands to the last state in the remembered sequence, and adds the new 'current' state to the end of the sequence. Undo simply removes the last state from the sequence (but never removes the first state).

$$
\begin{array}{lll}
doit^b : S^b \times C^b \to S^b & & \\
\text{where} \quad doit^b(hs, c) & = & hs \frown doit(last(hs), c) \quad c \neq undo \\
\quad doit^b(hs, undo) & = & chop_1(hs) \qquad\qquad length(hs) > 1 \\
\quad doit^b(< s >, undo) & = & < s >
\end{array}
$$

($chop_1$ is the function which removes the last item from a sequence.)

Surprisingly, its effective history function is simpler than that for the single-step undo:

$$
\begin{array}{lll}
eff^b : H^b \to H & & \\
\text{where} \quad eff^b(<>) & = & <> \\
\quad eff^b(h \frown c) & = & eff^b(h) \frown c \\
\quad eff^b(h \frown undo) & = & chop_1(eff^b(h))
\end{array}
$$

Again, the conservativeness condition is trivially maintained, but the encapsulation condition must be proven.

Note that, this time, the effect of an action on the effective history *can* be defined purely in terms of the effective history. The effect of undo truly is a function $H \to H$. So backtracking undo is less reflexive than single-step undo, which has to 'remember' that the last command was an undo. Of course, the simpler form of the effective history does *not* mean that it is simpler to implement, and as we discussed, the cost of storing enough to reconstruct all past states will be substantial. Also note that there are alternative equivalent formulations of the states of these two systems, and, most important, the implementations will usually involve storing 'deltas' information sufficient to reverse the effect of an action, rather than copies

of past states. Specifying the behaviour of a system and actually implementing it are very different! However, it is quite important during implementation to know *what* it is you are trying to implement, even if you have to work out *how* to achieve it for a particular system.

6 Adding redo

The raison d'être of undo is the user's need for a function that allows him to reverse the effect of a command, recovering a past situation. This is needed when the command has been performed in error so that an undesirable state is reached. What happens if the user realises that *undo* has been performed in error and has itself resulted in an undesirable state? The answer to this question has been the raison d'être of the redo function.

It is common to consider redo as simply the inverse of undo. Indeed, this may be the semantics of redo when undo is applied on a single action. But the meaning of redo is less clear in the case of multiple-undo. Is its effect the reconstruction of the last undone action, or of all the deleted history? And what about the effect of redo when the last undo has deleted a block of actions? Does it recover the whole block or only the last action in the block? Moreover, not only is it unclear what redo does, but there is also a complex dependence between it and undo. In the case of single-undo/single-action, undo is considered the inverse of the undone command and redo the inverse of undo. But in any semigroup we have that the inverse of the inverse of a function is the function itself. This means that the inverse of *Undo (type('x'))* is *type('x')*. But redo is not identical to *type('x')*, it is just that when performed at a *particular point of the history*, it has the same semantics. This has two consequences. Firstly, we could consider redo as a sort of super syntactic sugar. In principle, the user could simply repeat the undone command; redo just makes this easier (possibly substantially easier). We could say that the domain of interest of redo is not so much undo itself as the *undone action(s)*. Secondly, like undo, we have to consider at what level we expect redo to reverse the effect of undo. Certainly redo is not *simply* the inverse of undo!

After saying what redo *is not*, we need to progress to some definition of what it *is*, or at least, as we did with undo, explore the range of options.

When considering undo, four major issues arose: reflexivity, granularity (single or multiple action), repetition (single or multiple undo) and the idea of commitment points. Each of these has parallels for redo, and in addition the properties of redo are linked to those of undo. Although redo may not be a simple inverse of undo, it is intimately connected. We will see dependence when considering the granularity of redo, but also that there is an intrinsic dependence of causality that determines whether redo is meaningful.

6.1 Causality

Just as you cannot think of undo without considering what has been *done*, you cannot consider redo without something having been *undone*. This gives rise to the most basic property of redo:

causal dependence: in order to perform a redo, undo must have been performed.

This appears too obvious to bother stating, but serves to highlight the reflexive nature of redo. With undo, we had to consider whether the principal domain of definition is the

ordinary command history, or the action history itself. With redo, we move up a level: is it (i) simply the command history, (ii) the history with undo commands, or (iii) does it also know about its own role in the interaction? The causality condition would imply at least some knowledge at level (ii). So, its *effect* may be simply in terms of the undone actions, but it must at least know that they have been undone.

6.2 Granularity and repetition

Redo, like undo, may be applied to single or multiple actions. However, there is the additional issue of to what extent this is tied to the granularity of the undo command. We refer to such a linkage as *granularity dependence*. So, in addition to there being one or more candidate *undone* commands to redo, these may have arisen because of one or more actual *undo* commands. This gives rise to five kinds of potential redo granularity:

(a) redo of last undone action

(b) redo of some number of undone actions

(c) redo of all the actions undone by the last undo command

(d) redo of all the actions undone by some number of undo commands

(e) redo of all undone actions (up to the last non-undo command)

In this (c) and (d) exhibit granularity dependence, whereas (a), (b) and (e) only exhibit causal dependence (they redo *undone* actions).

The last (e) corresponds to a sort of escape, which reverses the effect of an entire sequence of undos. Similar escapes occur at the ordinary undo level; for example, many systems have a 'revert' menu option, which allows you to restore a document to the last saved version. Such escapes are themselves a sort of undo operation and are often considered in the same context [21]. Given that the effect of undo can be so confusing, such an escape from an undo dialogue may well be a good idea!

We can look at each of the undo categories from figure 7 and see how they interact with these kinds of redo granularity. Recall that class (iii), single-undo/multiple-action, was deemed an unreasonable alternative, so we will only consider the other three cases.

(i) *single-undo/single-action*: In this case, there can only ever be one undone action and one (effective) undo, so all five redo categories collapse into one.

(ii) *multiple-undo/single-action*: In this case undone commands and undo commands are in one-to-one correspondence, so (a)=(c) and (b)=(d). However, categories (b) and (d) look weird. If the system is going to allow single redo commands to have non-singular effects, why not allow this for undo?

(iv) *multiple-undo/multiple-action*: In this case, (a) is the weird option. If you can undo groups of actions, why only allow single redo steps? The same argument could be said to hold for (c) with respect to (d), but perhaps, given the different semantic level, one could argue that in some systems (c) may be more comprehensible than (d).

As with undo, we find that the granularity of redo interacts strongly with the possibility of repeated redos, but in addition it also interacts with the classes of undo. We can consider the remaining categories above and see which make sense when we consider single and multiple redo.

With case (i), multiple-redo is meaningless (only one thing to redo!), leaving us a single category of redo, flip-undo, where the undo and redo toggle between two states. As only one of undo or redo is possible at any time, the same button or menu position is used for each, leading to the apparent situation where undo is self-applicable. However, as we saw earlier, this 'undo of undo is redo' situation is never quite uniform between undo and other commands.

For both cases (ii) and (iv), the 'escape' redo can only be invoked (as a redo) a single time (although of course it might toggle, like flip-undo, undoing the redo!). Would one want such a redo in these circumstances? It might be argued for on efficiency grounds: a system may store only *backward deltas*; that is, information sufficient to undo commands, but not redo them. During a cycle of undoing, the system need only store the last not-undone state and the current state: the escape redo would simply jump back to this last not-undone state. However, although this is credible, the extra expense of two-way deltas over and above one-way deltas is not enormous and so it is likely that a redo of the 'escape' form would only be supplied in addition to more incremental redo.

In case (ii), we dismissed options (b) and (d), leaving only redo granularity (a/c) to consider. For reasons similar to those that ruled out undo of class (iii), we can also see that allowing only a single redo of granularity (a/c) would be silly. If we allow repeated undos, we need to have all the expense of machinery and memory to store lots of states, so why not allow multiple invocations of redo also? That is, we should only have options (a/c) with multiple redo, where each redo reconstructs more and more of the undone history of commands.

Finally, in case (iv), we have a similar story. Options (b), (c) and (d) only make sense for multiple redo, where they perform a similar job reconstructing the command history.

Figure 8 summarises this taxonomy. Note again the strong 'diagonal' of the table: granularity and repetition correlate, both within the operations of undo and redo, and between them. If you are going to go to all the trouble of storing lots of history information you might as well use it!

As we did for undo, we can summarise this in two informal alternative definitions:

Definition 3 *Redo is a system function which allows the user to recover the past state removed by* the *previous undo.*

Definition 4 *Redo is a system function that allows the user to recover a past state removed by* any *previous undo.*

Definition 3 corresponds to flip-undo. As with undo, there is a design choice between achieving defn. 4 by the user doing lots of redos (cases ii.a/c and iv.c), or with a single large granularity redo (cases iv.b and iv.d). Finally, the difference between (iv.b) and (iv.d) is in the interpretation of 'a past state' in definition 4, whether it is '*the* past state removed by any previous undo' or '*any* past state removed by any previous undo'.

Undo \ Redo	Single redo	Multiple redo
(i) Single-undo Single-action	(a)=(b)=(c)=(d)=(e) Redo only of the last undone action	Only one command to redo!
(ii) Multiple-undo Single-action	(a)=(c) Redo only of the last undo of a sequence (e) Redo as an 'escape'	(a)=(c) Redo of the last performed undo. Repeated redo used to reconstruct the history
(iv) Multiple-undo Multiple-action	(b/c/d) Single redo of a block of undone commands. Size may be different from the last undo. (e) Redo as an 'escape'	(b/c/d) Redo of a block of undone command. The size may be different from the last undo. Repeated redo used to reconstruct the history

Figure 8: A taxonomy of the redo function. The rows represent the kind of redo; the columns represent the kind of undo that may precede redo.

6.3 Reflexivity

As we saw in section 6.1, there is an inevitable reflexivity in the nature of redo – it cannot exist without reference to the previous occurrence of undo. In the ACS model this is captured in the state of the 'pending script'; however, for some kinds of redo (i.a) this is overkill, for others (iv.c/d) insufficient. The active script and pending script contain only the ordinary commands and so represent a base level of reflexivity: looking at the interaction with the underlying application. The more complex cases require the 'pending' script to record aspects of the complete action history – undo/redo are reflecting on their own behaviour.

For systems where the undo can be described purely in terms of the pending script, the redo and undo operations can be regarded as having a domain of the form $H \times H$, (active script $\times$ pending script). The flip-undo is a degenerate example, as the pending script never has more than one command (only one level of undo is allowed), and the causal dependence is captured by the fact that the pending script is only not empty if there has been a previous undo. Based on this, it is possible to fully describe flip-undo using three rules:

1. ordinary command – add it to active script and empty pending script
2. undo – if pending script is empty remove the last command from the active script and put it in the pending script
3. redo – if pending script is not empty remove the command from the pending script and add it to the active script

It is, of course, because the last two of these rules are disjoint that a single button or menu option can be used. Although this is a valid description of the behaviour, it is not how any such system is actually implemented – one wouldn't bother to store the whole active script and then never use it! Indeed, even for the formal specification, we will use just two copies of the state: current state and past state – similar to the single-step undo. With such a representation, both undo and redo simply swap the two states – identical! The system does not need to know whether it is doing an undo or a redo, the difference is in the user's interpretation of the effect. This is closer to the way it would be implemented.

The most common and straightforward kind of multi-step undo/redo can also be described using the basic ACS pending script. This is the policy found in Microsoft Word 6 and in the history list of Netscape Navigator. In these systems you can undo any number of commands one by one, or even undo several commands at once, using a menu. The behaviour can be described in a similar manner to flip-undo:

1. ordinary command – add it to the active script and empty the pending script
2. undo n – remove the last n commands from the active script and add them to the pending script
3. redo n – remove the last n commands from the pending script and add them to the active script

Notice that (as we saw with single-step and backtracking undo) the description is *simpler* (no conditions of the pending script) because it is more uniform, even though it is far more costly to implement. Word 6 and Netscape use different interface metaphors to represent these: in Word 6 the user has separate undo and redo menus, which exactly correspond to

the active and pending scripts, whereas in Netscape there is a single 'Go' menu with a tick against the currently displayed page. The Word 6 menus show commands (e.g. 'typing'), whereas in Navigator the items in the menu are pages, which correspond to states. The latter difference is a direct consequence of the more identifiable nature of the web browser state (a WWW page). Note also that the Netscape interface suggests a model that, rather than having two scripts, has just one script with a pointer $H \times Ptr$. This is equivalent to the $H \times H$ representation, but is in some ways more flexible (as we see below).

Not all undo systems can be described using a simple pending script. Systems of type (iv.c/d) need to have some record of how many commands were undone by a previous undo in order to redo them. The raw pending script merely records the list of undone commands, so such systems need a pending script that itself contains undo commands! In fact, it is easier to think in terms of a pointer into the complete action history; that is, undo/redo acting on a domain of the form $H^a \times Ptr$. All commands add something to the end of the history. Ordinary commands add themselves and set the pointer to the end. The undo/redo command takes the action currently pointed to, adds the inverse of the action to the end of the list and moves the pointer back one. Whether the undo/redo command is regarded as undo or redo is dependent on what sort of command is pointed to, and depending on how the inverses of commands are represented, the difference may be one of interpretation, rather than of different behaviour within the application. This sort of strong reflexivity sounds quite complex, and indeed in GNU Emacs, where it is used, no amount of experimentation seems to be able to uncover the rule! However, exactly the same rule is used in HyperCard's 'back' menu function (one of its two forms of history), and it seems less confusing there. This form of undo is rather like having your actions recorded by a video, which you can rewind to find previous states that you want to restore. However, the video keeps recording even when you are rewinding. Rewinding ordinary recording is undo, and rewinding past a previous rewinding is redo! Possibly a time or video-player metaphor would make such an undo/redo policy more comprehensible.

6.4 Summary of redo

We have reduced the space of possible redo mechanisms to three major categories (ignoring the escape type of undo), which can all be found in extant systems: flip-undo, non-reflexive multi-step undo/redo and reflexive multi-step undo/redo. These arose out of consideration of three of the four issues we introduced at the beginning of this section: granularity, repetition and reflexivity. The fourth issue, commitment points, we will return to in section 8.1. First, however, we will see how each of the three fit into the formal recovery framework.

7 Formal models of redo

Recall that we expected that any undo/redo system would form a conservative extension of its corresponding 'ordinary' system. That is, we are already part way to a general formal model of undo! Furthermore, the linear property (eqn. 5.4) in section 5.2 was also a generic property of any recovery system. So we have also for redo:

$$\forall h \in H_A \bullet \exists h' \leq h \textbf{ st. } eff(h \frown redo) = eff(h') \qquad \text{linearity of redo}$$

However, this simply says that after redo the system is in the same state as at *some* previous point in the interaction history. The informal definitions of redo are much stronger and we can give stronger generic formal conditions also.

As in section 5, we will first look at properties of undo corresponding to informal definitions and then look at models of specific redo policies. These policies will again be expressed as conservative encapsulations of an arbitrary underlying application.

7.1 Definition 3 – redo of *the* previous undo

We can formulate this one-step redo using algebraic properties similar to those for single-step undo. We want to formulate the idea that the state after undo followed by redo is the *same* as the state before the undo. As with undo, we can use stronger or weaker notions of sameness. The strong form is:

$$undo \frown redo \sim null \qquad (\textit{strong-ur})$$

That is, the *full* state after undo-redo is identical to that before either.

Recall that the equivalent *strong-cu* property only admitted pure backtracking undo with no possibility of redo – if you have forgotten about the previous command you cannot redo it! However, in the case of redo this is not a problem. Although such a property says that you have indeed 'forgotten' that the undo has ever happened, why do you need to remember it? There are many possible commands before the undo, so you want the *system* to remember which command was undone, and to redo the relevant one for you (what the redo function does!). However, with redo, there is only one thing it can 'undo' – the undo command itself. You can easily 'redo' this by doing another undo! This is, in fact, also a consequence of the general formal property that, in a semigroup, the inverse of an inverse is the original command. The *strong-ur* condition says that the inverse of undo is redo, and as a consequence the inverse of redo is undo:

$$redo \frown undo \sim null \qquad (\textit{strong-ru})$$

Flip-undo satisfies this strong undo property. At first glance, it looks as though this is also true for non-reflexive multi-step redo, as found in Word 6 and Netscape, at least for single-action undo/redo commands. It is almost true, but fails at the 'end points' when the active or pending scripts are exhausted – this will be clear in the specification below. In contrast, reflexive multi-step undo, as found in GNU Emacs and HyperCard 'back', is not even close to satisfying it! Such systems remember that there has been an undo-redo pair and subsequent undoing will reveal this to-and-fro-ing in the interaction history. In being truly reflective on their own interaction, such systems can forget *nothing* and so *never* return to a previous state.

Note that as with the similar undo property, the property says 'and there can be only one redo'. However, the only case where single redo was useful was flip-undo, where there is only one thing to redo anyway!

7.2 Definition 4 – redo of *some* previous undo

This is like the linear undo property, except that there we simply said that recovery commands return to some previous state or effective history. For redo, we want to say that this corresponds to the situation immediately prior to some previous undo:

$$\forall h \in H_A \bullet \exists h' \in H_A \ \textbf{st.} \ \begin{array}{l} h' \frown undo \leq h \\ \wedge\ eff(h \frown redo) = eff(h') \end{array}$$

This says that the redo restores all the effect of the undo. For the weaker case where we allow redoing of part of a block of undone commands, we need to weaken the last part of the above to:

$$eff(h') \leq eff(h \frown redo)$$

That is, the effective history is somewhere between that immediately prior to the undo, and that just after. For the case where each undo applies to only one command, the two properties are identical.

7.3 Specific redo policies

Flip-undo

Flip-undo turns out to be a simpler version of single-step undo. As we noted earlier, it can be regarded as two states between which the system flips back and forth when the undo/redo button is pressed. The projection function again peels off the second part of the state:

$S^f = S \times S$
$s_0^f =< s_0, s_0 >$
$proj^f : S^f \rightarrow S$
where $proj^f(< p, s >) = s$

As undo and redo can use the same button or menu option, the command set is simply the ordinary commands plus the undo button.

$C^f = C \cup \{undo\}$
$doit^f : S^f \times C^f \rightarrow S^f$
where $doit^f(< p, s >, c) = < s, doit(s, c) > \quad c \neq undo$
$doit^f(< p, s >, undo) = < s, p >$

Compare this with the definition of single-step undo in section 5.4. The definitions for the $doit$ function are simpler (only two cases rather than three); this is also true of implementations where one is storing the current state and some sort of delta. It is easier always to store something in the delta, either to go backward or forward, than to sometimes store nothing. Also compare the two cases for $doit^f$. See how the second (undo) case does not lose information. This means that a perfect redo is possible using a second undo function. That is, this undo satisfies the strong-uu property (equivalent to strong-ur as undo/redo are one action):

$$undo \frown undo \sim null$$

However, imagine what happens after a command-undo pair. Suppose the system starts off in state $< p, s >$. After the command c, this becomes $< s, s' >$ where $s' = doit^f(s, c)$. A subsequent undo changes this to $< s', s >$. Note that this is *not* identical to the state before the command. That is, flip-undo does not satisfy the strong-cu property. However, the projected part is identical, so it does satisfy weak-cu:

$c \frown undo \sim_{proj} null$

The effective history function is also similar to that of single-step undo, but is again slightly simpler with fewer special cases.

$$
\begin{array}{lll}
eff^f : H^f \rightarrow H & & \\
\text{where} \quad eff^f(<>) & = & <> \\
\qquad\quad eff^f(h \frown c) & = & eff^f(h) \frown c \quad c \neq undo \\
\qquad\quad eff^f(undo) & = & <> \\
\qquad\quad eff^f(h \frown a \frown undo) & = & eff^f(h) \qquad a \in C \cup \{undo\}
\end{array}
$$

Non-reflexive multi-step undo/redo

As we noted earlier, this could be expressed using a pending script and an active script, or a command history and a pointer into it ($H \times Nat$). However, instead we will define it using a sequence of states as we did for backtrack undo, but with a pointer to allow redo.

$S^m = S^+ \times Nat$
$s_0^m = << s_0 >, 1 >$
$proj^m : S^m \rightarrow S$
where $proj^m(< hs, n >) = hs[n]$

Remember that this is *not* how one would implement such a system – an actual implementation would use deltas. The intention is to specify behaviour not style of construction!

The undo command decrements the pointer and the redo command increments it (until it reaches the beginning or end of the script). This moves the current state, the state picked up by $proj$, backward and forward. Ordinary commands chop the history back to this current state and then act upon it.

$$
\begin{array}{lll}
C^m = C \cup \{undo, redo\} & & \\
doit^m : S^m \times C^m \rightarrow S^m & & \\
\text{where} \quad doit^m(< hs, n >, c) & = & < hs[1 \ldots n] \frown doit(hs[n], c), n+1 > \\
\qquad\quad doit^m(< hs, 1 >, undo) & = & < hs, 1 > \\
\qquad\quad doit^m(< hs, n >, undo) & = & < hs, n-1 > \qquad n > 1 \\
\qquad\quad doit^m(< hs, n >, redo) & = & < hs, n > \qquad n = length(h) \\
\qquad\quad doit^m(< hs, n >, redo) & = & < hs, n+1 > \qquad n < length(h)
\end{array}
$$

We can see now that this does not satisfy the strong-ur property: in the case when the state is $< hs, 1 >$, undo leaves this as $< hs, 1 >$, which when followed by an redo gives $< hs, 2 >$, which is not what we started with. If we distinguish effectual and non-effectual undos and redos, we could say that this form of undo/redo satisfies strong-ur and strong-ru when the first command of the pair is effectual.

The effective history can be defined recursively where undos remove preceding commands and redos remove preceding undos! The structure is most clearly expresde using the composition of the existing effective history for backtrack undo eff^b and a new function eff^R, the definition of which looks remarkably similar to eff^b itself:

$eff^m : H^m \rightarrow H$
where $eff^m = eff^b \circ eff^R$

$eff^R : H^m \rightarrow H^b (= (C \cup \{undo\})^*)$
where $eff^R(<>) = <>$
$eff^R(h \frown c) = eff^m(h) \frown c \quad c \in C$
$eff^R(h \frown undo) = eff^m(h) \frown undo$
$eff^R(h \frown redo) = chopUndo(eff^R(h))$

The subsidiary function $chop_{undo}$ is like a limited version of $chop_1$, it removes the last action from a history, but only if it is an undo:

$chop_{undo} : H^b \rightarrow H^b$
where $chop_{undo}(<>) = <>$
$chop_{undo}(h \frown c) = h \frown c \quad c \in C$
$chop_{undo}(h \frown undo) = h$

Notice that eff^R is the identity on histories which contain no redos. This reflects the fact that if you never use the redo command, this system is identical to backtrack undo. However, the structure of the function eff^R suggests something more. The non-reflexive multi-step undo/redo is itself a conservative encapsulation of the backtrack undo! In fact, if we modified the definition of backtrack undo to allow some of the command set to be non-undoable, we would find that the redo part is exactly a second level of backtracking undo built upon the original backtrack undo!

Reflexive multi-step undo/redo

In this last case, we will again be able to have a single undo/redo command, the interpretation of which (as undo or redo) depends on context. That is, the command set is:

$C^r = C \cup \{undo\}$

The undo system is aware of its own full interaction history (i.e., including the effects of undo itself) and the state of the system must reflect this. Again, we could do this by using a history and pointer as the state. However, this time, we would have needed $H^r \times Nat$, the complete action history. In fact, it is simpler to use a state sequence with pointer, *identical* to the state of non-reflexive multi-step undo/redo.

$S^r = S^+ \times Nat$
$s_0^r = << s_0 >, 1 >$
$proj^r : S^r \rightarrow S$
where $proj^r(< hs, n >) = hs[n]$

However, although the state may be identical, the *doit* function is different! This time every command, including undo, adds to the end of the state sequence, with the sole exception of an undo when undo gets back to the beginning.

$doit^r : S^r \times C^r \rightarrow S^r$
where $doit^r(< hs, n >, c) = < hs \frown doit(hs[n], c), n + 1 >$
$doit^r(< hs, 1 >, undo) = < hs, 1 >$
$doit^r(< hs, n >, undo) = < hs \frown hs[n-1], n - 1 >$

Again we can define the effective history recursively with the help of subsidiary funtion U_i^r. Basically it says that if you want to work out the effective history of an action history ending in n undoes, you chop off the preceding n actions and then work out the effective history of what is left.

$$
\begin{array}{lll}
eff^r : H^r \to H & & \\
\text{where} \quad eff^r(<>) & = & <> \\
\qquad\quad eff^r(h \frown c) & = & eff^r(h) \frown c \\
\qquad\quad eff^r(h \frown undo) & = & eff^r(U_1^r(h))
\end{array}
$$

$$
\begin{array}{lll}
U_i^r : H^m \to H^m & & \\
\text{where} \quad U_i^r(<>) & = & <> \\
\qquad\quad U_i^r(h \frown c) & = & chop_i(h \frown c) \quad c \in C \\
\qquad\quad U_i^r(h \frown undo) & = & U_{i+1}^r(h)
\end{array}
$$

Notice especially the fact that the both the domain and range of U_i^r is the full action history H^m. This is the reflection in the formal model of the deeply reflexive nature of this undo.

8 Understanding redo

8.1 Commitment

Suppose you are using a system with flip undo. When you have entered a command a you could still use undo to remove the effect of that command. That is, you are not yet *committed* to the command a. However, as soon as you enter any other command b this command implicitly commits the previous command – with only one step of undo you can no longer retract the command a. In a similar way ordinary commands commit the undo action. If you enter the commands '$a\,undo$', the undo is not committed because with flip undo a subsequent undo will reverse the effect and a will be reinstated. However, if you enter '$a\,undo\,b$' the command b *commits* the undo – it is no longer possible to change your mind.

Commitment points are those moments during interaction when some decision become irrevocable. Without undo every command is its own commitment point. With undo this is no longer the case, but as we have seen commitment points do occur.

Consider now the case of non-reflexive multi-step undo/redo. Now when the user enters a command it does not commit previous commands as the user can perform multiple undoes. Also the undos are not committed as they can be revoked with a redo. Even the redos are not committed as you can simply do more undos. Does this mean there are no commitment points with this sort of undo? In fact, there are commitment points – when the user first performs and ordinary command after a series of undo/redo commands that commits the undo/redo sequence. Suppose that we have entered this sequence of actions:

$c_1, c_2, c_3, undo, c_4$

At the command c_4 we commit the $undo$. This forms a branching point in the history of the system: from the main 'trunk' of interaction c_1, c_2 we have the initial branch c_1, c_2, c_3 and the new branch c_1, c_2, c_4. Reflexive multi-step undo/redo remembers all these branches (at a cost of being rather complex and confusing), that is, it has no commitment points. However,

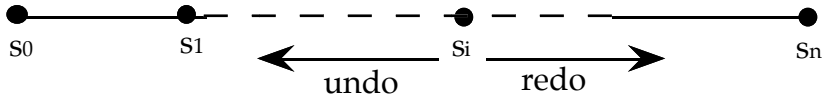

Figure 9: The linearity of the history: undo is the past in the past, redo is the future in the past.

non-reflexive multi-step undo/redo forgets the original branch once a new branch starts to form. *During* a sequence of undo/redos the current state pointer moves backwards and forwards through the history, but as soon as anything else happens, the history is chopped off, rather like the stack in backtrack undo. We can see this situation as a subdialogue during the human-computer interaction and we call it *undo mode*. Outside of this subdialogue, the whole sequence of undo/redo actions during undo mode can be treated as a single undo of a block of actions without redo. For example, suppose that we have entered this sequence of actions:

$c_1, c_2, c_3, c_4, c_5, U(3), R(2), U(1), c_6$

where U(i) indicates that we have performed an undo of the last i commands; similarly for the redo. Outside the undo mode, this sequence is effectively seen as:

$c_1, c_2, c_3, c_4, c_5, U(2), c_6$

which is in turn equivalent to c_1, c_2, c_3, c_6.

Examples of systems of this class include Netscape and Word 6. Indeed, this is the model which is followed by most modern systems which implement multi-step undo (but not necessarily all history systems).

8.2 Visualising undo

Systems make decisions on the smallest granularity of undo: which low-level actions in the form of keystrokes, menu selections, dialogue-box interaction are considered one undo-able command. In the case of single undo/redo the user gets no choice on the granularity they have one step of whatever size the system decides. However, in the case of multi-step undo/redo, it is the user who chooses which state he is interested in reaching.

If the user can undo (or redo) to different points in the history, not just the last/next state then, as a consequence, a visual representation is required of the recovery functionality: not simply buttons or icons, but lists. In many systems this is represented as a menu of past actions which can be undone. However another approach is taken by some Visual Query Systems. For example Hypercube [5] simultaneously displays all the past states of the query formation interaction as windows. Undo is then simply choosing a window. However, the Hypercube technique cannot be applied to a text editor, because the large number of states which may belong to the history of a document!

8.3 Undo/redo: browsing the past

In Section 1 we introduced undo as a special case of reachability, as it allows the user to reach a past state. So what about redo?

If we confine ourselves to non-reflexive multi-step undo, then redo also can be seen as a form of reachability. Consider what happens *during* an undo mode. Let's suppose that, starting from the initial state s_0 we have reached, the state s_n performing only commands. At this point the sequence of past states is $\{s_0, s_1, ..., s_n\}$. Suppose also that by performing a multiple undo we have reached the state s_i. Now, s_i is the current state and the sequence of states which undo allows us to reach is $\{s_0, s_1, ..., s_i\}$. So undo continues to be a function of the past, allowing the user to go in the left side of the past history (fig. 9). In the current state, we can say that undo is the past of the past. Instead, the set of states which redo allows to reach is $\{s_i, s_{i+1}, ..., s_n\}$. From the current state s_i redo is a function of the future, allowing the user to go in the right side of the past history. So also redo is a special case of reachability, allowing the user to reach a future state in the past.

9 Final analysis

Undo can be very confusing to users, at just the moment when they have done something wrong and need help from the system. This is partly because there are several models of undo and in this report we have laid out the possibilities, both as taxonomies of undo and redo and as formal specifications of the range of possibilities.

This formalisation process has been systematised by an abstract formal framework, the conservative encapsulation, which captures the way in which the original behaviour is preserved within the extended system with undo/redo functionality.

Undo is complicated because it is reflexive, both in that it looks at the process of the interaction and even sometimes undo itself! This reflexivity is reflected in its formalisation and models of undo need to deal with two levels of state: that of the simple system as if it had no undo, and that of the full system with undo. The full state of the system usually involves histories either of commands or states.

The form of undo/redo which is commonly found in recent systems is non-reflexive multi-step undo/redo. The apparent power of the undo/redo mechanism is limited by the commitment points which occur when ordinary commands terminate period of undo/redo activity. Premature commitment to actions is the risk that undo helps avoid. The fact that this commitment resurfaces can therefore be the cause of more risk. This is an issue which we pursue further elsewhere [11].

Whereas reflexive multi-step undo/redo does not have these commitment points its highly reflexive nature makes it very difficult to comprehend. However, this is not only found in Emacs, a tool of the dedicated computer scientist, but also in HyperCard. Whether users of hypertext browsers can comprehend such a navigation mechanism would make an interesting empirical study.

Within the undo subdialogues of non-reflexive undo/redo we have seen an interesting phenomena whereby undo/redo move the users idea of the 'current state' backwards towards the relative past and forwards to the relative future, browsing through time for the required state in the absolute past.

References

[1] G.D. Abowd and A.J. Dix. Giving undo attention. *Interacting with Computers*, 4(3):317–342, 1992.

[2] J.E. Archer, R. Conway, and F.B. Schneider. User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages and Systems*, (6):1–19, 1984.

[3] T. Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Transactions on Human Computer Interaction*, (1):269–294, 1994.

[4] T. Berlage and M. Spenke. The GINA interaction recorder. In J. Larson and C. Unger, editors, *Engineering for Human-Computer Interaction*. North-Holland, 1992.

[5] S.K. Chang, M.F Costabile, and S. Levialdi. Reality bites – progressive querying and result visualization in logical and vr spaces. In *IEEE Symposium on Visual Languages*, pages 100–109. IEEE Computer Society Press, 1994.

[6] Rajiv Choudhary and Prasun Dewan. A general multi-user undo/redo model. In Hans Marmolin, Yngve Sundblad, and Kjeld Schmidt, editors, *ECSCW'95*, pages 231–246. Kluwer Acadamic, 1995.

[7] A. J. Dix and C. Runciman. Abstract models of interactive systems. In P. Johnson and S. Cook, editors, *People and Computers: Designing the Interface – Proceedings of HCI'85*, pages 13–22. Cambridge University Press, 1985.

[8] A.J. Dix. *Formal Methods and Interactive Systems: Principles and Practice*. D.phil. thesis, ycst 88/08, Department of Computer Science, University of York, 1987.

[9] A.J. Dix. *Formal Methods for Interactive Systems*. Academic Press, 1991.

[10] Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human–Computer Interaction*. Prentice Hall, 1993.

[11] Alan Dix, Roberta Mancini, and Stefano Levialdi. Alas I am undone – reducing the risk of interaction? In *HCI'96 Adjunct Proceedings*, page 51–56, 1996.

[12] Alan J. Dix. Moving between contexts. In P. Palanque and R. Bastide, editors, *Design, Specification and Verification of Interactive Systems '95*, pages 149–173. Springer Wien, 1995.

[13] G.B. Leeman. A formal approach to undo operations in programming languages. *ACM Transactions on Programming Languages and Systems*, 8(1):50–87, 1986.

[14] R. Mancini. Modelling interactive computing exploiting the undo draft PhD thesis, University of Rome "La Sapienza", 1996.

[15] B. A. Myers and D. S. Kosbie. Reusable hierarchical command objects. In *Proceedings of CHI 96, Vancouver, BC, Canada*, pages 260–267. ACM Press, 1996.

[16] A. Monk P. Wright and M. Harrison. State, display an undo: a study of consistency in display based interaction. Technical report, University of York, 1992.

[17] B. Shneiderman. The future of interactive systems and the emergence of direct manipulation. *Behaviour and Information Technology*, 1(3):237–256, 1982.

[18] R. M. Stallman. EMACS: The extensible, customizable self-documenting display editor. *ACM SIGPLAN Notices*, 16(6):147–156, 1981.

[19] H.W. Thimbleby. *User Interface Design*. Addison Wesley, 1990.

[20] J.S. Vitter. US&R: A new framework for redoing. *IEEE Software*, 1(4):39–52, 1984.

[21] Y. Yang. Undo support models. *International Journal of Man-Machine Studies*, (28):457–481, 1988.

Appendix A

In section 3.5, we said that the right hand side of 'the cube' could derived from the rest. That is, we can prove that the state conservativeness diagram, figure 3, commutes if figures 2 and 4, both commute.

In fact, one needs a slight caveat to this statement. Depending on how one formulates the state of a system, it may include so called 'garbage' states. That is states which are permitted by the description, but can never occur in a real system. For example, if the position of the cursor in a text editor is represented by an integer denoting the offset into the text, then it will always point to a position within the text. However, a simple definition of the state as:

$$S = Text \times Int$$

would in principle include unreachable states such as $<$ "hello", 500 $>$, where the cursor points to a location outside the text. Such a state would *never* be reached during normal use of the system and is thus 'garbage' in the formal description. We do not want, nor need to say anything about the properties of such states, they never happen and are not interesting.

Given the above, we will regard the diagram in figure 3 as referring to all *reachable* states in S^a and S. This means that we can assume for any state s in S^a, there is a corresponding action history h from H^a which gives rise to s (i.e. $s = I^a(h)$). We will see that this 'weak reachability' is necessary for the proof.

We will consider the two parts of figure 3, the left triangle and the main square separately. Looking first at the triangle this says that the initial states must agree. That is we want to prove that:

$$proj(s_0^a) = s_0 \qquad \text{** to prove}$$

we prove this as follows:

$$\begin{array}{rcll} proj(s_0^a) & = & proj(I^a(<>) & \text{defn. of } I^a \\ & = & I(eff(<>)) & \text{figure 2} \\ & = & I(<>) & \text{figure 4} \\ & = & s_0 & \text{defn. of } I \text{ (figure 6)} \end{array}$$

□

The commutativity of the square corresponds to proving that:

$$proj(doit^a(s, c)) = doit(proj(s), c) \qquad \text{** to prove}$$

which is the statement of the appropriate state update when ordinary commands are used. To prove this, we need the weak reachability property and choose h_s such that:

$$s = I^a(h_s) \qquad \textit{weak reachability}$$

Given this definition we proceed as follows:

$$\begin{array}{rcll} proj(doit^a(s, c)) & = & proj(doit^a(I^a(h_s), c)) & \text{defn. of } h_s \\ & = & proj(I^a(h_s \frown c)) & \text{defn. of } I^a \\ & = & I(eff(h_s \frown c)) & \text{figure 2} \\ & = & I(eff(h_s) \frown c) & \text{figure 4} \\ & = & doit(I(eff(h_s)), c) & \text{defn. of } I \\ & = & doit(proj(I^a(h_s)), c) & \text{figure 2} \\ & = & doit(proj(s), c) & \text{defn. of } h_s \end{array}$$

□