The cube – extending systems for undo

Alan Dix[†], Roberta Mancini[‡] and Stefano Levialdi[‡]

[†] School of Computing, Staffordshire University,
PO Box 334, Beaconside, Stafford ST18 0DG, UK.

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

Abstract When a system is extended by adding undo, the original system behaviour should be preserved within the new extended system. In this paper a formal framework is established which considers the state of the system before and after the extension and captures the relationship between the layers in a structure we call *conservative encapsulation* or *the cube*. Alternative algebraic properties of undo and examples of two specific undo policies are formalised within this framework. The framework afforded by conservative encapsulation is not just suitable for undo, but can be used to study other forms of system extension such as history mechanisms.

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

1 Introduction

1.1 The problem

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?

This paper examines the way interactive systems are extended. We establish a formal framework which we call *conservative encapsulation* for formulating such extensions. The *encapsulation* part captures the relationship between the original system and the new extended system. The *conservativeness* condition formalises the notion that the original system behaviour is still preserved within the extended

system. This is named in a simiar vein to the use of the word 'conservative' in the study of the semantics of formal notations.

This formal framework considerably expands our expressive power for talking about undo and related problems. Previous work on undo has largely started with specific policies and then proved properties of those policies. We are now in the position where we can take properties and derive the policies.

The focus of this paper is on undo and redo, but similar issues arise when behaviour is extended in other ways. Indeed, the same framework has already been used to study the related issue of history mechanisms in hypertext systems and world-wide web browsers [9].

1.2 Undo – linear and non-linear

The issue of undo in user interfaces has been studied by several authors over many years (e.g. [2, 15, 25, 22, 18]). Indeed, the ability to undo (not necessarily with an undo button) is seen as one of the key features of the direct manipulation paradigm [20].

However, undo is far from a 'solved' problem. Experiments have shown that even experienced users of word prcessors find it difficult to work out what undo will do [24]. Also in our own work we have discovered that the most popular form of multi-step undo, backtrack undo (see below), can add an unexpected risk of loosing work – instead of reducing risk in interaction it can add to it [10]!

In most single-user undo systems, there is a single stream of activity and the undo command simply reverses the effect of the most recent command. This is linear undo. Much of the recent work on undo support has focused on different kinds of non-linear undo. One focus is on selective undo where the user is allowed to undo commands other than the last one. A recent example of this is GINA [3], but in fact this issue goes back many years [23]. The other recent focus is on multi-user undo [1, 4, 5]. This is similar to selective undo in that a user may want to undo their own last action, which may not be the last action in the entire system trace. In both cases the meaning of subsequent commands may need to be altered to suit the altered context of use [12].

In this paper we will stick to examples of the 'simple' case of linear single-user undo. In fact we believe that the framework we propose applies equally to selective undo, but the issues for single-user linear undo are sufficiently interesting to investigate in their own right.

1.3 Backtrack undo

Many recent systems include a specific form of multi-step undo. During interaction a record is kept of all commands to-date. When the user performs the undo command (by menu or undo-button) the effect of the last ordinary command is reversed. Multiple invocations of undo take the system further and further into the past. When a normal command is eventually performed the system forgets all the undone commands and continues as if they had never been executed. We call such an undo policy (with no redo) a *pure backtrack undo*.

Such systems also typically have a redo command. As the undo commands are executed the undone ordinary commands are added to a list, sometimes called the 'pending' list [2]. If after one or more undo commands the redo command is executed, the last command from the 'pending' list is reinstated, reversing the effect of the previous undo. Thus undo and redo act rather like cursor keys moving the current state back and forth through the history of recent interaction. However, the first ordinary command wipes out the pending script effectively *committing* the sequence of undo/redo commands. We will refer to this later as *stack-based undo/redo* as it is rather like pushing and popping elements from a stack.

This form of undo policy, is found in Microsoft Word 6 and most other recent applications. Also the history mechanism in Netscape Navigator behaves in a similar fashion. We will use the two forms of backtrack undo as our examples later in the paper.

This is not the only undo policy found in popular systems. Older systems often only allow a single level of undo/redo – a policy called flip undo [22] because the system toggles back and forth between the most recent two states. Also users of EMACS [21] will have experienced a far more complex form of multi-step undo, which we will not even attempt to describe here!

1.4 Formal models of undo

Many different formal models have been proposed in the literature in order to describe undo in interactive systems [2, 23, 15, 22, 1, 11]. Indeed, most papers describing implementation mechanisms or other features of undo use some sort of formalisation, although differing greatly in the degree of rigour. Normally the purpose of such formalisations has been to investigate a specific policy or mechanism.

In this paper we produce a fully abstract framework for understanding undo, within which the algebraic properties of different policies can be posed and compared. The principle focus of this paper is on the framework as a way of capturing the idea of extending a system. However, its use as a foundation for comparing the formal properties of different undo policies is explored in greater detail elsewhere [16, 17].

By 'fully abstract' we mean that the model we present is in a sense canonical for any model of undo within the domain of single-user closed systems. Whereas previous work has specified undo for specific systems or specific undo policies or implementation mechanisms, this framework, the cube, captures the essence of what any undo policy must satisfy. This is rather like the difference between specifying particular sorting algorithms (quick sort, bubble sort, etc.) as opposed to specifying the principle of sorting. However, this analogy is not quite right as there is effectively only one sorting function with several ways to implement it, whereas there are many kinds of undo and the framework needs to be rich enough to include them all.

1.5 Structure of the paper

In the next section we will look further at the problems that arise because of the reflexive nature of undo and how this forces us to consider two levels of 'state'

within a system with undo. Section 3 formalises these two levels of state and most importantly formalises the relationship between them in the *conservative extension*. In fact, the power of the conservative extension is that it is not just about undo but about extending behaviour in general. It tells us how pre-existing behaviour is preserved, but not what undo actually does! Focusing on undo, section 4 looks at how different algebraic properties of undo and redo can be formalised within the framework. Finally in section 5 this is all exemplified as we formulate two particular undo policies: backtrack undo with and without redo.

2 Tale of two systems

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 [7], 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* [25]. 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 [25]. In fact, it is shown [7, 8] 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

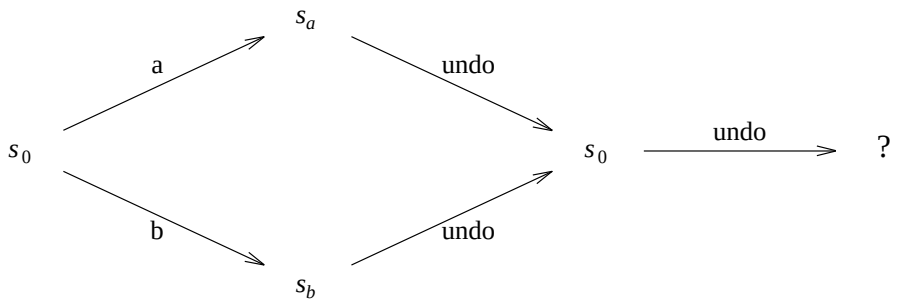

Fig. 1 Undo of undo?

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 Mac users cite Word 5, which both use the flip-undo policy. In each case, the undo command toggles back and forth between two states of the system. 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 state of the system if there were no undo whatsoever. That is the state of the original system before we ever thought of adding 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.

This distinction between the two levels of states may seem obvious to some readers, but the authors' experience of presenting aspects of undo formally and informally suggests that it is one of the most difficult concepts to grasp. Indeed, problems of interpreting state are also evident in many other aspects of user interface specification.

First note that the distinction between S and S^a is *not* the distinction between the internal state of the system and the external visible state (as manifest in the visual, tactile or auditory display). The state/display relationship is important and is an issue that has been studied extensively in previous work [14, 7]. The two states we are dealing with in this paper are in fact the internal state of the original system and the internal state of the system when undo has been added. The original system will have its own display, as will the system with undo. These two displays may be identical if undo is executed by a keyboard command only, but more often the adding of undo will also influence the user interface, with extra menu options, on-screen buttons, etc. The effective design of the user interface for undo is certainly not a simple problem, but in this paper we will confine ourselves to the different kinds of internal state.

Second, the choice of S^a in interpreting figure 1 is not arbitrary. The reader may think that if a different interpretation of state were chosen, then the proof would fall apart – strong-cu and strong-uu could be consistent. In fact, this is not the case. The full state S^a is the only *real* state of the system – all other interpretations of state are precisely that, abstractions and interpretations of the real full state S^a. The full state S^a is precisely *all* the knowledge and stored information within the system that affects its behaviour with respect to the outside world (including the user). There is no other candidate that has this unique status.

More problematic is S. This has the same privileged status if we were interacting with the version of the system *before* the code for undo was added. Its status is far less clear when we look at the complete system. It is not a 'state' in a formal sense, in that its evolution cannot be defined in terms of its current state and the external influences on it, but is at most a projection of the full state. Yet this is precisely the 'state' that a user (including most computer scientists!) may think of when they are interacting with the complete system, even whilst using undo, and it is why they believe vi or Word 5 satisfies strong-cu. So, the 'state' in a user's own mental

model of the system is *not* the real state of the system, neither is it clear that it has an objective definition at all. It is exactly the capturing of the relationship between the real state of the system S^a and the commonly perceived state S that is the subject of this paper.

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 [25]. 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 all possible 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 $C^u = (C \cup \{undo\})$.

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 ($H = C^*$). Again in the case of a single undo command, this simplifies to $H^u = (C \cup \{undo\})^*$.

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.

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.

These two kinds of command could be seen as operating on the two kinds of state respectively. However, is to some extent and over simplification as the ordinary commands must have some effect on the full system state (including command history etc.). It is precisely this which we will capture using *conservativeness* in the next section.

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.

Recall the example with which we started this paper. Your company wanted undo added to a word processor, but instead you ended up with an undoable spreadsheet. Somehow we want to say that this is wrong. To do this 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 [6], using multiple levels of abstraction as found in [7].

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

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

[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.

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.

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

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:

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

The projection function would then be:

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

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 $doit^a$:

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

The equivalence '$\sim$' often gives a more compact and algebraic formulation 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.

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:

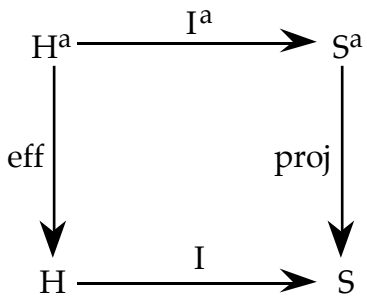

Fig. 2 Encapsulation.

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

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

(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.

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

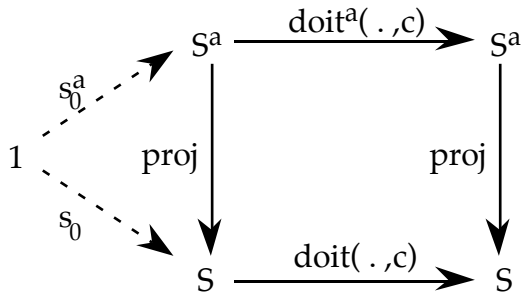

Fig. 3 Conservativeness of state.

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.

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

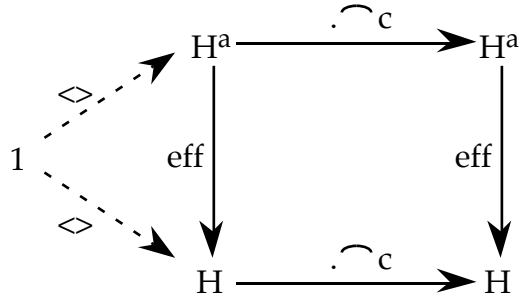

Fig. 4 Conservativeness of effective history.

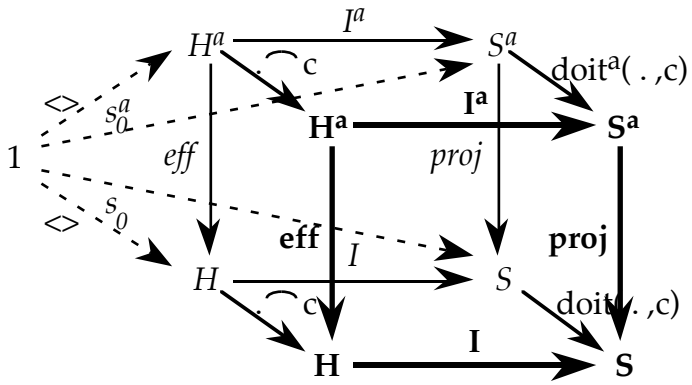

Fig. 5 The cube.

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 of the cube is similar, except that it refers to the full system.

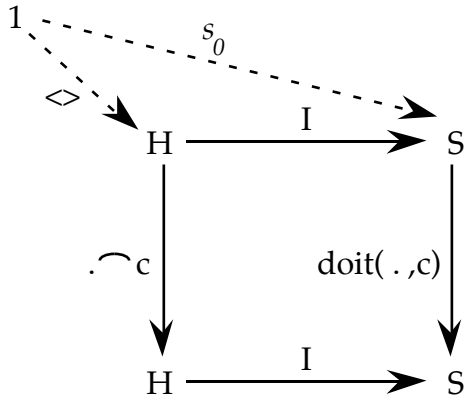

Fig. 6 Bottom of the cube.

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. 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.[2]

3.6 Summary of framework

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. So, 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.

[2] In fact, one needs a slight caveat to this statement. Depending on how one formulates the state of a system, it may include 'garbage' (or unreachable) states, that is states which are permitted by the description, but can never occur in a real system. To be precise encapsulation and conservativeness of the effective history implies conservativeness over all reachable states. However, this is sufficient as garbage states never really happen!

4 Properties of undo and redo

We have not, so far, considered the effects of undo commands themselves. Before looking at specific undo policies we will consider possible properties we may require of undo and redo commands.

4.1 Strong and weak undo

In section 2 we began with a definition that the state of the system after undo is the same as it was before the previous command. However, we now know how careful we must be with the interpretation of the word 'state'. One option is to consider the *full* state of the system as it was before the last command:

$$doit^{a*}(s, c \frown undo) = s$$

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

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

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 *strong-cu* exactly captures the specific case where undo can be used as a pure backtracking tool with *no redo*, and in [16] 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.

A weaker undo property can be obtained by focusing on the 'normal' component of the state, rather than full state:

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

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

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

Note that this is similar to *strong-cu*, but only says that the projection onto the original component of the state is restored. It is thus far weaker than *strong-cu*, both intuitively and in the formal sense that if *strong-cu* is true then *weak-cu* follows.

4.2 Redo too

We can formulate redo using algebraic properties similar to those for 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 we said 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 undo, so the *system* must remember which command was undone in order 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 the stack-based undo/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.

5 Examples of undo

The implementation of an undo system will typically be very dependent on the underlying application. However, policies such as flip undo or backtrack undo, exist independently of the particular application. Thus we can 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 give specifications of backtrack undo and stack-based multi-step undo/redo in just such a manner. We will use the superscript b for elements in the model of backtracking undo (e.g. S^b for the full state) and m for elements of model of stack-based multi-step undo/redo

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]).

In both cases, the full state of the system will include multiple copies of the original system state (S) as both models must include history of past interaction.

5.1 Backtracking undo

In the case of backtrack undo, the full command set consists solely of the original commands plus the special undo command. We will use superscripts b for the elements of this model. So we have:

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

With backtracking undo all the past states (in the sense of original state) 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).

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

($chop_1$ 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:

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

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.

5.2 Stack-based multi-step undo/redo

We will define this form of undo/redo behaviour using a sequence of states as we did for backtrack undo, but with a pointer to allow redo.

$S^m = S^+ \times Nat$
$s_0^m = << s_0 >, 1 >$
$proj^m : S^m \to S$
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.

$C^m = C \cup \{undo, redo\}$
$doit^m : S^m \times C^m \to S^m$

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

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 expressed 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:

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

$eff^R : H^m \to H^b (= (C \cup \{undo\})^*)$

$$
\begin{array}{lll}
\text{where } eff^R(<>) & = & <> \\
eff^R(h \frown c) & = & eff^R(h) \frown c \qquad c \in C \\
eff^R(h \frown undo) & = & eff^R(h) \frown undo \\
eff^R(h \frown redo) & = & chop_{undo}(eff^R(h))
\end{array}
$$

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:

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

5.3 Layers upon layers

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 stack-based 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!

We can verify this relationship by constructing the corresponding projection function:

$$
\begin{array}{l}
proj^R : S^m \rightarrow S^b \\
\text{where } proj^m(< hs, n >) = hs[1 \ldots n]
\end{array}
$$

This definition of $proj^R$ makes the two layers of conservative encapsulation commute in figure 7.

6 Summary

We have seen how systems with undo must be considered at two levels of abstraction: considering the state of the sytem with and without undo. We have defined *conservative encapsulation*, a formal framework based on the PIE model, which captures the appropriate relationship between these levels so that the behaviour of the original system without undo is preserved within the full system with undo.

This paper confined itself to single-user linear undo. However, various forms of non-linear undo are increasingly found in research sytems. The current framework is already suitable for studying selective undo, but the algebraic properties may be less elegant than for linear undo. It cannot without modification handle multi-user undo.

The power and significance of this framework is that it gives a unified context within which specific undo policies can be defined and that it allows the general reasoning about properties of undo independent of the particular policy implementing the policies. In contrast previous work on formalising undo has generally specified specific polices and then proved properties of those policies, the exception to this being the proof repeated here in section 2.1. Most important the framework formally captures what we expect to be true, that the system behaves 'sensibly' for non-undo commands, but which is at best tacit and certainly not formally justified in all previous work of which the authors are aware.

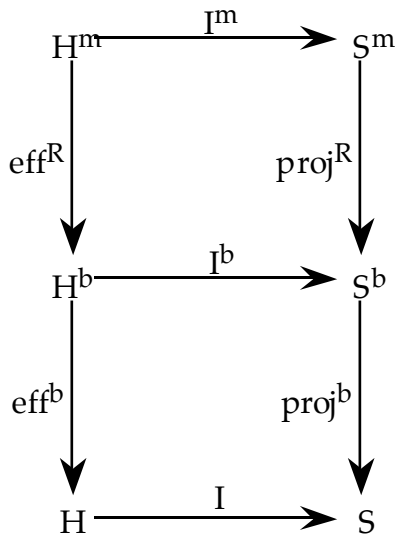

Fig. 7 Two layers of conservative encapsulation.

We have seen two examples of undo policies formulated within the cube framework: pure backtrack undo and stack-based multi-step undo. Each can be expressed as a conservative encapsulation of the original system. Furthermore, we have seen that the second of these is itself a conservative encapulation of the first. That is, the pure backtrack undo is extended by adding redo to give stack-based multi-step undo/redo, but the behaviour of the original pure backtrack undo is preserved within the larger system.

So, within this paper we have seen that conservative encapsulation can be applied both to undo as an extension of an underlying non-undoable system and to redo as an extension of undo. Also in related work we have found the very same framework can be used to analyse history mechanisms [9]. Thus conservative encapulation is not only an abstract framework for understanding undo, but also a more general starting point to consider many kinds of system extension.

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] 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.

[6] 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.

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

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

[9] Alan Dix and Roberta Mancini. Specifying history and backtracking mechanisms. In Fabio Paterno and Phillipe Palanque, editors, *Formal Methods in Human–Computer Interaction*. Springer-Verlag, 1997.

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

[11] Alan J. Dix. Dynamic pointers and threads. *Collaborative Computing*, 1(3):191–216, 1995.

[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] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*. Springer-Verlag, 1985.

[14] M.D. Harrison and A.J. Dix. Modelling the relationship between state and display in interactive systems. In P. Gornay and M. J. Tauber, editors, *Visualisation in Human–Computer Interaction*, volume LNCS 439, pages 241–249. Springer-Verlag, 1990.

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

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

[17] Roberta Mancini, Alan Dix, and Stefano Levialdi. Formal and informal definitions of undo. Technical Report RR9611, University of Huddersfield, 1996.

[18] 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.

[19] D.T. Sannella. Semantics, implementation and pragmatics of Clear, a program specification language. Technical Report CST-17-82, PhD thesis, University of Edinburgh, 1982.

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

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

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

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

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

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

Appendix: Relationship with other types of refinement

One of the reviewers commented on the relationship between the work described in this paper and the notions of refinement in the wider formal methods community. The notation, naming and types of refinement vary greatly between different parts of this community so it is impossible to give a complete picture, but we will briefly consider some of the links here.

In [7] various forms of morphism between PIEs are discussed. The encapsulation in figure 2 is a form of 1-morphism. In this and other earlier work on relations between PIEs the mapping between histories (labelled P for 'program') was usually referred to as '$parse$' as it corresponds to a translation between input languages. However, the principal use of the 1-morphism was to represent refinement or abstraction relationships, for example the relationship between the functional core of an application and the full system including its user interface. In such cases the

$parse$ function was expected to be monotonic in its argument (derived history always grows). In contrast, the very nature of undo is that the value of $eff(h)$ function may reduce when undo commands are added to h. The name '$proj$' (projection) for the mapping between states (effects) has been retained as it is a standard mathematical projection between two sets. Note that we have called the particular 1-morphism of interest here 'encapsulation'; this is because the word 'extension' was already used in this early work to refer to a different kind of morphism. Also it is important to note that the PIEs used in this paper are all 'monotone' in the sense that their 'effect' part is a state. The study of PIE morphisms was itself inspired partly by the elegant categorical semantics of algebraic specification [13, 19] (sadly never equalled in the model-based specification world) and, most importantly, by homomorphisms of classical mathematical structures such as groups.

Another related area is the modelling of state–display conformance [14]. The similarity is not surprising, as the information for the current display must be contained within the whole state of the systems. Hence the display is an abstraction of the system state just as the underlying state S is an abstraction of the full state with undo S^a. The state–display conformance is precisely a refinement relationship as found in many formalisms: each concrete state has a corresponding abstract state (display) and each concrete operation (command) on the system state has a corresponding abstract operation (command) on the display. This work was designed to address the consistency and visibility of changes to the system as manifest at the display. However, whereas there was an effectively one-to-one mapping between abstract commands and concrete commands in the case of state–display conformance, in contrast the whole importance of undo is that it extends the command repertoire. The work on state–display conformance has diagrams equivalent to figure 3, except that the state–display diagrams hold for all commands (abstract and concrete). In contrast, figures 3 and 4 are only meaningful for the original command set, not for any additional undo commands.

Consider now refinement in a general formal methods sense. There are two levels at which we can formulate the cube within such a framework but each only give us part of the picture.

First we can treat a PIE as a single-sorted algebra with sort S. The initial state s_0 is considered a constant operator and each command from C is considered a separate unary operator. This is similar to the specifications one often sees in Z where each command is specified using a separate schema. In this formulation, figure 3 corresponds to a refinement relationship with $proj$ being the retrieve function mapping the 'concrete' state S^a onto the 'abstract' state S. Although this does capture one aspect of the relationship it is not enough in itself. The extra operations (undo, etc.) in the augmented command set C^a suggest that we ought to consider it more as a Σ-algebra enrichment than a refinement, as occurs during the development of system specifications from simpler building blocks. Indeed, this is exactly why the term 'conservative' is used for this diagram as it corresponds to the conservativeness condition demanded of certain enrichments: the new sorts and operations can add to the existing functionality, but not alter that of the pre-existing Σ-algebra. However, even this is not quite sufficient as we have in fact simultaneously refined the state. Another problem with the formulation within a single-sorted algebra is that

the role of eff is difficult to capture. This function maps not between sorts of the algebra, but between histories of commands. If the commands are considered as operations, then eff is effectively a mapping between the *formulae* of the algebras. This is precisely why the term $parse$ was used in early formulations of PIE morphisms!

A second formulation is as a multi-sorted algebra where the set of histories H and the state S are sorts and the operations include the constant s_0 and the binary operation $doit$. Note that this is a 'lifted' formulation as the commands which were considered as operations in the first formulation are considered as sort elements here. In this formulation we can again consider the cube as a form of refinement where $proj$ is again the retrieval function from S^a to S and eff is the retrieval function from H^a to H. In this formulation, figure 2 is exactly the standard refinement obligation. However, this time we have great difficulty in encompassing the conservativeness condition. It is possible to add C as a sort with operations to make H a free semigroup over C. However, the corresponding retrieval function from C^a to C is *partial* because C^a is an extension of C.

So the first formulation has trouble with the encapsulation condition and the second, 'lifted' formulation has trouble with the conservativeness condition. The problem is that conservativeness is talking about the effects of individual commands whereas encapsulation is talking about the relationship between sequences of commands. They are talking at different levels. This is not surprising given the intention is to have a framework which is sufficiently rich to describe the inherently reflexive nature of undo! We can see this clearly in the properties of eff. For refinement we would normally expect eff to be a one-to-one mapping (concrete commands to abstract). If, instead, we consider a component to whole relationship, we would expect eff to be monotonic (as you do more things to the state as a whole, more things happen to the components). However, to allow the reflexive nature of undo the eff function may be both many-to-one and non-monotonic.

As we have seen, the nature of conservative encapsulation is related to various types of refinement, but has its own special nature. Over the years the specification of interactive systems has highlighted issues for the wider formal methods community. One point to note in this light is the importance of reachability in proving the sufficiency of conservativeness of history (footnote in section 3.5). If we did not restrict ourselves to reachable states, the result would *not* hold. The importance of reachability is often overlooked. This has parallels in the refinement of abstract data types and of objects. It is commonly assumed that a condition of refinement is that pre-conditions are progressively weakened and post-conditions are always strengthened. In fact, it is also possible to have valid refinements of ADTs where pre-conditions are actually strengthened. This is because the full set of ADT states will typically not be reachable. The 'reachability' effectively forms a hidden invariant which is strengthened as post-conditions are strengthened and which can then be harvested in order to strengthen pre-conditions. Every programmer knows this, you can assume values of ADTs are well behaved because you know how they were created! However, it is surprising just how often this is forgotten when the ADTs are treated within a formal framework.