Dynamic Pointers and Threads

Alan Dix

August 30, 1993

1 Introduction

The notion of location is central to virtually all manipulation. For a single-user system there is typically one key point of reference, often reified in the shape of an insertion point or mouse pointer, but even then there may be subsidiary locations, such as marked blocks or hyper-text links. For group working the issue becomes more problematic, with different users having different views of shared objects, and operating at different locations.

This paper concerns *dynamic pointers* – a mechanism for the description and implementation of various forms of location information, and *threads* – light-weight versions expressing the divergence due to the distributed nature of much group working. For the purpose of this paper, threads will be seen as the subsidiary concept, although one could discuss threads in their own right.

The domain which will be addressed is group editing/update, but drawn widely to encompasses virtually all manipulation of shared information.

The remainder of this section will give a brief introduction to dynamic pointers and threads. Then in the next two sections we shall consider dynamic pointers in more detail. First a rationale as to why pointers are not just one small detail in system design, but, instead, one of the key issues. After that we will consider a model for dynamic pointers – an semi-formal description of a formal model described in more detail elsewhere [2]. Thus far, the material will be largely a review of previous work on dynamic pointers targeted at single-user interaction.

Sections 4–6 will discuss the application of dynamic pointers to various forms of group editing. First of all we will consider synchronous editing, how dynamic pointers can describe the appropriate behaviour of multiple insertion points, and has also been used to analyse the various options for group undo support. Then Section 5 will show how Ellis and Gibbs' algorithm for distributed synchronous group editing can be seen as a special case of dynamic pointers. In the course of this discussion, we will also introduce the concept of threads. Finally in Section 6, we will see how dynamic pointers and threads apply to more widely distributed asynchronous editing.

During the discussion we will see that dynamic pointers are pervasive in existing systems, but largely implicitly. In the last part of this chapter (Section 7), we will see that dynamic pointers are not only a way of talking about implementations, but are an effective implementation device.

Dynamic pointers

Dynamic pointers were originally developed several years ago to deal with issues of location (cursors etc.) in single-user systems [2]. What then are they doing in a book about CSCW – group working? Well, the context in which dynamic pointers were developed was single-users with *multiple views*. It is not surprising then that they are readily applicable to various problems in groupware. In particular, we shall see that Ellis and Gibbs' algorithm for distributed synchronous group editing can be regarded as a 'special case' of dynamic pointers.

Dynamic pointers describe the update behaviour of various forms of location information such as text insertion points, selections, other block markers and various forms of hyper-text anchor

Imagine two users, Adonis and Beatrice. They are working using a shared editor and their current document reads:

```
Adonis␣is␣↑␣and␣Beatrice␣is␣↓.
```

The sentence is partial and both users are about to type in their prime personal characteristic in order to complete it. Adonis' insertion point is denoted by the up arrow (↑) and Beatrice's insertion point is the down arrow (↓). Beatrice types first yielding:

```
Adonis␣is␣↑␣and␣Beatrice␣is␣beautiful↓.
```

Adonis then types "adorable", but unfortunately the implementor of the group editor was not very expert and the resulting display was:

```
Adonis␣is␣adorable↑␣and␣Beatrice␣is b↓eautiful.
```

Beatrice's insertion point followed the 36th character before Adonis' insertion, and followed the 36th character after. Reasonable but wrong. The actual text should clearly read:

```
Adonis␣is␣adorable↑␣and␣Beatrice␣is␣beautiful↓.
```

Of course, no designer would make this mistake – or at least if they made it they would soon notice and correct it. But are they sure their solution works in all cases? They need some higher level understanding of the phenomena than that given by scenarios or by the program code itself.

Figure 1: Pointer problems in synchronous group editing

point. Most important, we are able to discuss the behaviour of multiple pointers into the same object. This is clearly useful if we are to deal with several users' insertion points while engaged in group editing (see Figure 1).

In addition, many interfaces can be seen as composed of several layers, with the actual application objects at the deepest level, with various levels of viewing and presentation. For a multi-user interface, users may well be accessing the same shared objects, but be doing so via very different views. Dynamic pointers also help us to express the relationship between interface layers, and thus make it easier to distinguish those user actions which are truly shared from thus which concern the individual user's interface.

Dynamic pointers serve several rôles.

- They are a *descriptive* mechanism for aspects of the users' (shared) interface.
- They are a *specification* mechanism for defining the structure and behaviour of the software system.
- They are an *implementation* mechanism which can be used as an underlying architecture for both centralised and distributed systems.

We will see examples of each of these uses throughout this chapter. Dynamic pointers will be used to describe the desired behaviour of synchronous group editing (as in Figure 1). They will then be used to discuss Ellis and Gibbs' algorithm for distributed synchronous group editing, that is, in a specification rôle. Finally, at the end of the paper, we will see how dynamic pointers are an effective implementation mechanism, and in particular, make for truly extensible group widgets.

Most of the examples in this paper will concern linear text. Dynamic pointers apply equally to non-linear structures, such as trees or hyper-text. However, sequential text turns out to be the most difficult case.

Threads

As soon as we have several users working at a distance on the same objects there are potential conflicts due to several users attempting to update the same object at the same time. At very least some users' views may be out-of-date. the traditional approach to this problem is to prevent it occurring, by locking and mutual exclusion. However, in many cooperative working situations this is not suitable. Threads are a way of describing the resulting divergence of users' views of an object and ways of attempting reconciliation.

Threads can be thought of as light-weight versions and are named by analogy with the 'threads' or light-weight processes, found in several operating systems and programming environments. The light-weightness can be extreme – in local network applications, the divergence between different users' versions may only last for a few seconds.

Threads are closely connected with dynamic pointers, as the latter form a mechanism to facilitate the reconciliation of differing versions. Like dynamic pointers their use can be notional – as they describe the interface phenomena and algorithms, or practical – as an explicit feature of the implementation.

2 Rationale

Although the example in Figure 1 shows that dynamic pointers are a real phenomenon, are they important enough to warrant a whole book chapter? In this section we shall see that the behaviour of pointers is not just one interesting issue in the implementation of interactive systems, but one of the key issues. It is fundamental to direct manipulation and is pervasive as an implicit concept in the implementation and behaviour of many systems. Further argument as to the importance and uses of dynamic pointers can be found in [2].

Direct manipulation

Most modern interfaces are to some extent 'direct manipulation'. This is not only an aspect of their physical appearance but influences the way in which objects are denoted. Consider a query to a bibliographic database:

```
SELECT name, title FROM bib
    WHERE name = "Dix"
```

The mode of reference here is *descriptive*. That is, the object of interest is referred to by some aspect of its value.

Contrast this with file selection in a graphical interface such as the MAC finder – the user simply points the mouse at the file's icon and clicks the mouse button. One can think of this as "what you can see is what you can grab". The mode of reference is *indicative*, by pointing rather than naming.

So, issues of location and pointers are not just important to direct manipulation, but they are germane to its very nature.

Implementation state

If we look within an interactive system, a similar story emerges. What data structures do we find? First of all there are the objects of interest which are being manipulated: text in a word processor, a spreadsheet or database records. Then there are flags and other simple data types used to record the mode of interaction or dialogue stage: text changed or not, currently selected window, contents of current menu. As well as the objects of interest themselves, there are often sub-objects, for example, in cut/paste buffers, or copies of the objects kept for history purposes. Finally, we find pointers *into* the objects of interest: currently selected spreadsheet cell, offset of window into text, set of currently selected database records.

In addition, it is the pointers which have the most important role when mapping between actions at the presentation level (such as mouse clicks on screen objects) and those on the underlying objects. As we have seen this is the fundamental mode of interaction in direct manipulation and thus all such interfaces must have some representation of these pointer-pointer mappings.

Existing support

Despite their obvious importance, issues of location and pointers are rarely made explicit. The pointers are there in real systems, as they must be to represent cursor positions, marked blocks etc., but are treated in an *ad hoc* manner resulting in inconsistency and unnecessary complexity. The specific 'dynamic' nature of the pointers may often be simply an artefact of the implementation rather than part of the design. In a few cases dynamic pointer-like facilities are offered to the programmer or user, but they are not integrated into the tools as a whole, for example, the built-in cursor is treated as a different category. The author's own work on the specification and implementation of dynamic pointers is the only attempt of which he is aware to deal with them in an explicit and integrated fashion.

Probably the only reason this situation can persist is that many systems have relatively few explicit pointers. In particular, most applications admit only one view onto an object, thus there is at most one point at which update occurs. This situation is likely to change, even for single user systems, as operating systems support live links between applications thus opening up the possibility of multiple simultaneous views.

Groupware

For groupware systems, multiple views and multiple insertion points are normal. In addition, the separation between the presentation layer and the underlying objects is not simply an architectural abstraction, but central to the participant's model of interaction. — different users interact with different parts and different representations of the same shared object. Furthermore, such systems are often implemented over distributed platforms and the *ad hoc* solutions which were acceptable for single-user systems cannot be relied upon.

In the light of this, the need to deal explicitly with dynamic pointers can no longer be seen as an academic exercise, but is essential to any practical implementation. Given this, it is not surprising that the only published algorithm for synchronous distributed text editing [5] is very close to the author's own work on dynamic pointers. It is also interesting to note that the only implemented versions of multiparty undo are where the domain is structured rather than free text — precisely those domains where dynamic pointers (*ad hoc* or explicit) cause fewest problems. We will see in Section 4 that dynamic pointers can be used to address the problem of group undo.

3 A model of dynamic pointers

This section describes a formal model of dynamic pointers. Further details of this model can again be found in [2].

Objects, pointers and operations

We start of with the set of objects being manipulated, and call the set Obj. Examples, of this would be text (sequences of characters) or trees. In fact, the most difficult case is sequential data such as text where insertions are allowed in the middle.

$$\begin{array}{lcll} Obj & = & \mathrm{seq}\, Char & - \text{ text} \\ Obj & = & (root : Node, left, right : Node \twoheadrightarrow Node) & - \text{ trees} \end{array}$$

As well as the objects we obviously want to talk about the pointers into those objects. We call the set of such pointers Pt. For text the pointers could be represented by an integer, representing the offset into the text. For example, the pointer 2 into the text '`abcd`', would be pointing to the gap between the `b` and the `c`.$^d agger$ [1] For trees, a pointer might be regarded as a sequence of Ls and Rs, representing the path from the root.

$$\begin{array}{lcll} Pt & = & \mathbf{Nat} & - \text{ text} \\ Pt & = & \mathrm{seq}\{L \mid R\} & - \text{ trees} \end{array}$$

The nature of such representations is not of primary importance, it is the existence of the pointers which is crucial. Indeed, the actual implementation of pointers may well involve direct references to the data structures storing the objects, rather than being a simple index. It is worth noting, that the examples above are simple in that they both represent *atomic* pointers to specific points of the structures (although the tree pointers could refer to interior nodes as well as leaves). In addition, one needs to consider *block* pointers such as those referring to a region of text. In a sequential structure such pointers may be represented as a pair of atomic pointers to the start and end, but this is not the only possibility. Another complication is that not all pointers are valid for all objects. For example, the pointer 7 is not valid in the text '`abcd`'. The issues of block pointers and valid pointers are dealt with in detail in [2].

To manipulate the objects there are a set of operations such that each operation op takes some arguments and the old state of an object and gives us the new state of the object:

$$op : \quad Args \times Obj \longrightarrow Obj$$

The particular arguments depend on the operation, for example, an insertion in text would require the characters to be inserted and the location at which they are to be inserted. A 'clear all' operation would probably require no arguments at all. Notice that the arguments to the insert operation involve a pointer, e.g.:

$$\text{‘}\mathtt{abxcd}\text{’} = insert(\text{‘}\mathtt{x}\text{’}, 2, \text{‘}\mathtt{abcd}\text{’})$$

$$\text{‘}\mathtt{abxcd}\text{’} \quad = \quad insert(\text{‘}\mathtt{x}\text{’}, 2, \text{‘}\mathtt{abcd}\text{’}) \quad - \quad \text{insert } x \text{ at position } 2$$

This is frequently the case as many operations at the programmer's interface as well as at the user interface involve positions.

[1] † An alternative would be to have pointers to the characters themselves, so that 2 points to the character `b`.

Pull functions

Finally, we need to represent the effect of the update on the pointers. This is done by adding an additional 'pull' function for each operation. The pull function can either be regarded as a separate function with the same arguments as the original operation:

$$pull_{op} : \quad Args \times Obj \times Pt \nrightarrow Pt$$

or alternatively as a second result of the original operation:

$$op : \quad Args \times Obj \rightarrow Obj \times (Pt \nrightarrow Pt)$$

Usually the operation and its arguments will be implicit from context, so we will simply write *pull*, but remember that this is shorthand.

As an example of a pull function, consider the operation $insert(s, n, txt)$, where s is the string to be inserted, txt is the text into which it is inserted and p is the position for the insertion. Its associated pull function would be:

$$pull(pt) = \begin{cases} pt & \text{if } pt < n \\ pt + len(s) & \text{if } pt > n \end{cases}$$

That is, pointers after the insertion point n are shunted on by the length of the inserted string.

Note that the case when $pt = n$ has been deliberately left out. It is not clear what should happen in this case. Should pointers at the point of insertion end up at the end or beginning of the inserted string? We can think of the two posibilities as the 'push 'em along' strategy and the 'leave 'em behind' strategy. we can denote these two options for *pull* as $pull_+$ and $pull_-$. These are equal (to the above definition) when $pt \neq n$, but when $pt = n$ they have the following behaviours:

$$\begin{array}{lcll} pull_+(pt) & = & pt + len(s) & - \text{ push 'em along} \\ pull_-(pt) & = & pt & - \text{ leave 'em behind} \end{array}$$

The experience of cursors in single user editors suggests that the 'push 'em along' strategy is correct. This means that even the cursor causing the insertion can be treated exactly the same as all other pointers. However, we will see that the multi-user situation contradicts this and we find that the 'leave 'em behind' strategy is better. In this case, when an insertion is done using a cursor, that cursor must be explicitly (and atomically) incremented. However, this is not a great burden so long as all other pointers are treated uniformly (including other people's cursors). Furthermore, it is not too strange that the cursor which caused the update behaves slightly differently.

We will take the former case for now ($pull_+$) and then see in the next section how this leads to trouble.

Problems for pull functions

We have already seen one problem highlighted by pull functions: what to do with pointers at the point of insertion? Note that this problem is *not* caused by the representation. If we have additional pointers as well as the cursor, for example hypertext links, we need to address this problem. Unfortunately it is usually the programmer who has to deal with this as the designer has simply missed the issue. The dynamic pointer representation has therefore focused attention at the design stage.

There are other problems highlighted by dynamic pointers. For tree operations, we must consider what happens to pointers to nodes which have been deleted, whether the pointers are moved to point to the closest remaining ancestor node, or whether they become 'undefined'. Again a design not a solely implementation issue.

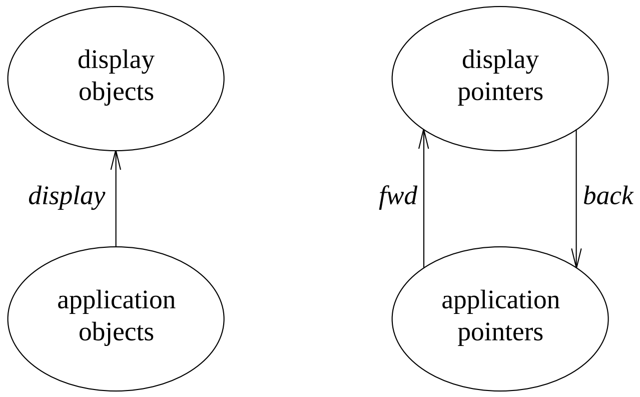

Figure 2: Mappings between levels in the interface

When we move from atomic pointers to block pointers, the problems become markedly worse. What is the appropriate 'pulled' value of a block pointer when an overlapping block is moved? There is no easy answer to such problems, but they should definitely be considered as part of the design process.

Further properties of pointers

Two further aspects of dynamic pointers are worth mentioning before we move one.

Firstly, we need to talk about the mappings between interface layers, in particular between the presentation and the underlying objects. In general, the displayed appearance is some function (*disp*) of the application objects (and other attributes of the interaction state). We need to be able to translate a pointer to the application objects (say a hypertext link point or a cursor) into the corresponding screen position. We also need to be able to translate screen positions (where a mouse is clicked) into application object pointers. We can represent these two mappings as functions *fwd* and *back* between the pointers at the two levels (Figure 2).

The *fwd* and *back* maps have similarities with the pull functions considered earlier. Whereas the pull function operates through time at the same level, the *fwd* and *back* functions operate at a particular moment between levels.

The ability to be able to deal with the presentation separately from the underlying application is very important for multi-user applications. However, there are no fundamental issues raised over and above those for single-user interfaces.

The second issue which is relevant to multi-user applications is the use of pointers to represent the *locality of change* of an update. For any operation, we can identify a block pointer representing that part which has been affected by the update. This can be used in various ways, for example, if we apply the *fwd* mapping we can determine possibly changed areas of the screen over which to apply 'repair'. In a single-user application such a mechanism might be seen as rather heavy as the location of change is usually obvious. However, where the change has originated with another user on a different machine, a generic mechanism to deal with such repair is very attractive. In addition, the block representing the changed areas can be used to detect possible interference between different operations, an issue which will become important later when we discuss distributed editing.

4 Synchronous group editing

The following three sections deal with group editing where the users' systems are progressively less tightly knit. This section deals with the simplest case: synchronous group editing where the network communications are fast enough for a centralised implementation. That is, we assume that there is no perceivable lag between the user pressing a key (or mouse action) and the feedback. Furthermore, the *feedthrough* is equally fast – that is, the time it takes for other users' displays to reflect my changes.

In implementation terms this probably means the main application runs on a central machine and minimal user interfaces on each participants' workstation relay events to the central application. The application then processes the events in the order they arrive and broadcasts changes to each participant. For example, this may be achieved by running the application as a single X client and opening windows on each participants' X server. In modelling terms, this means that we treat the application as a single state with users' actions modifying that state. In either case, we are deliberately excluding the possibility of race conditions.

Applying dynamic pointers

Look again at Figure 1. We can consider the state of the system to consist of the current text T and the two participants' insertion points ap and bp, which are pointers in the text. Now just before Adonis types 'adorable', Beatrice's insertion pointer is just after the 36th character ($bp = 36$). The operation is $insert(\text{`}\texttt{adorable}\text{'}, 10)$, where 10 is the current position of Adonis' insertion point ap. The corresponding pull function is:

$$pull(pt) = \begin{cases} pt & \text{if } pt < 10 \\ pt + 8 & \text{otherwise.} \end{cases}$$

The number 8 in this formula is the length of the insertion 'adorable'. If we then apply this to both insertion points we get $pull(ap) = 18$ and $pull(bp) = 42$, which are precisely the positions we want them at.

We can generalise the algorithm, assume we have a set of users U and for each user u there is an associated insertion point $u.pt$. Now, if a particular user v performs the operation op with associated pull function $pull$, the new state of the system is given by:

$$\begin{array}{rcll} T' & = & op(T) & \\ u.pt' & = & pull(u.pt) & \quad \forall\, u \in U \end{array}$$

The undecorated components (T and $u.pt$) represent the state before the operation, and the decorated components (T' and $u.pt'$) represent the state after. Figure 3 shows this algorithm applied to the scenario from Figure 1.

As well as updating all the users' cursors the pull function will update other pointers in the state such as the offset of the users' windows into the document.

Cursor wars

Consider the (nasty) case when two participants' cursors are at the same location. If we use the pull function as described above (obeying the 'push 'em along' stragtegy), then as one user types both cursors move together. Imagine that both Adonis and Beatrice type at the same time. The text starts out as:

$$\boxed{\texttt{Adonis␣is␣}\uparrow\downarrow\texttt{␣and␣Beatrice␣is␣.}}$$

They both simultaneously type in their idea of Adonis chief characteristic:

We begin with the initial state and display:

$$\begin{aligned}
T_0 &= \text{`}\texttt{Adonis␣is␣␣and␣Beatrice␣is␣.}\text{'}\\
ap_0 &= 10\\
bp_0 &= 27\\
display_0 &= \boxed{\texttt{Adonis␣is␣}\uparrow\texttt{␣and␣Beatrice␣is␣}\downarrow\texttt{.}}
\end{aligned}$$

Beatrice inserts the word 'beautiful', this gives rise to the following operation and pull function:

$$\begin{aligned}
op_1 &= insert(\text{`}\texttt{beautiful}\text{'}, 27)\\
pull_1(pt) &= \begin{cases} pt & \text{if } pt < 27\\ pt + 9 & \text{otherwise.}\end{cases}
\end{aligned}$$

The numbers 27 and 9 are the current value of bp (where the insertion takes place), and the length of the inserted material ('beautiful'), respectively. After this operation the text is updated and the pointers are operated on by the pull function:

$$\begin{aligned}
T_1 &= op_1(T_0) &&= \text{`}\texttt{Adonis␣is␣␣and␣Beatrice␣is␣beautiful.}\text{'}\\
ap_1 &= pull_1(10) &&= 10\\
bp_1 &= pull_1(27) &&= 36\\
display_1 &= \boxed{\texttt{Adonis␣is␣}\uparrow\texttt{␣and␣Beatrice␣is␣beautiful}\downarrow\texttt{.}}
\end{aligned}$$

Then Adonis types his contribution:

$$\begin{aligned}
op_2 &= insert(\text{`}\texttt{adorable}\text{'}, 10)\\
pull_2(pt) &= \begin{cases} pt & \text{if } pt < 10\\ pt + 8 & \text{otherwise.}\end{cases}
\end{aligned}$$

Leading to the final state of the system.

$$\begin{aligned}
T_2 &= op_2(T_1) &&= \text{`}\texttt{Adonis␣is␣adorable␣and␣Beatrice␣is␣beautiful.}\text{'}\\
ap_2 &= pull_2(10) &&= 18\\
bp_2 &= pull_2(36) &&= 44\\
display_2 &= \boxed{\texttt{Adonis␣is␣adorable}\uparrow\texttt{␣and␣Beatrice␣is␣beautiful}\downarrow\texttt{.}}
\end{aligned}$$

That is, exactly as we wanted!
There is one slight complication. In reality the insertions would be character by character. However, if you follow through the steps, you find that this makes no difference.

Figure 3: Synchronous editing with dynamic pointers

$$\boxed{\texttt{Adonis\textvisiblespace is\textvisiblespace adaobraysblmael}\uparrow\downarrow\texttt{\textvisiblespace and\textvisiblespace Beatrice\textvisiblespace is\textvisiblespace .}}$$

Of course (and happily) neither of their contributions is readable.

Imagine instead, we adopted the 'leave 'em behind' strategy for the pull function. If we simply left both cursors where they were then the typing would come out backwards! Obviously, and reasonably, one would treat the actual pointer which was the subject of the update differently and increment it, leaving all other pointers behind. In this case, imagine that Adonis starts to type fractionally before Beatrice:

$$\boxed{\texttt{Adonis\textvisiblespace is\textvisiblespace}\downarrow\texttt{ad}\uparrow\texttt{\textvisiblespace and\textvisiblespace Beatrice\textvisiblespace is\textvisiblespace .}}$$

This time the cursors have separated due to the insertion of the first two characters '`ad`'. From now on Adonis and Beatrice are typing at different locations so their typing will not become mixed:

$$\boxed{\texttt{Adonis\textvisiblespace is\textvisiblespace abysmal}\downarrow\texttt{adorable}\uparrow\texttt{\textvisiblespace and\textvisiblespace Beatrice\textvisiblespace is\textvisiblespace .}}$$

Clearly collaboration is not always best served by mutual inteligibility!

The above example suggests that the 'leave 'em behind' strategy is appropriate for pull functions in multi-user situations. However, we need to ammend our algorithm so that the cursor of the user who performs an update is treated differently. Assume as before that the update is done by user v, then the rule is:

$$\begin{aligned} T' &= op(T) \\ v.pt' &= pull_+(v.pt) \\ u.pt' &= pull_-(u.pt) \qquad \forall\, u \neq v \end{aligned}$$

Undo

Imagine we are the situation at the end of Figure 3 and Beatrice hits her 'undo' key. What should happen? There are two possibilities:

global – Adonis' insertion is removed, the last action by anyone

local – Beatrice's insertion is removed, her own last action

The global option is easiest to implement, but it most likely that Beatrice meant her own insertion, the local option. In a previous paper [1] it was shown that local undo can be achieved, when the users' actions commute, by reordering history and then performing a global undo.

For example, imagine Beatrice has performed the action b and then Adonis has performed the action b. That is the history of actions performed is ba. Now Beatrice hits the undo button. If a and b commute then the effect of ba is exactly the same as that of ab. So, we can pretend that the actions happened in the other order (ab) and so can perform a global undo on this state. We thus undo Beatrice's last action — local undo.

If when we swop the order of the actions we operate on any pointers using the respective pull functions, we find that most simple text based actions commute, that is rather than simply ab we get $a_b\ b_a$ where:

$$\begin{aligned} a_b &= pull_b^{-1}(a) \\ b_a &= pull_{a_b}(b) \end{aligned}$$

Note that we are slightly misusing notation here: $pull_b^{-1}(a)$ means the operation obtained when $pull_b^{-1}$ is applied to each pointer in a.

Note also, that having worked out the operation b_{undo} which would undo b_a, we can apply this directly to the state obtained as a result of the original operations ba. This is because this state is exactly the same as the rewritten history. So, we don't actually have to rewrite history, just act as if we have!

Figure 4 shows the effect of this approach on the running example. Although it looks slightly complex, each stage is simple enough and the process is the same no matter what actions are performed.

In fact, there is a slight complication in that it is not always the case that $a_b \; b_a$ has exactly the same effect as ba. The times where this fails are precisely those where the interpretation of local undo is suspect. For example, when the cursors overlap. In these special cases slight adjustments must be made or the undo should fail. For single character operations, there are sensible adjustments (corresponding to alternating the use of 'leave 'em behind' and 'push 'em along' pull functions). For block operations, the times when commutativity fails are when there is no sensible interpretation of undo at all and thus failing the undo is appropriate.

In all cases, we are not simply thrown back on our imagination to work out the problem situations. The formula $a_b \; b_a = ba$ can be simply checked (and it is not usually very complicated) and the cases where it fails are precisely those we need to consider for special treatment. The whole checking cycle is automatic (albeit requiring some mathematical manipulation).

5 Decentralised synchronous editing

We now move on to the decentralised condition. We assume that each participant is working on a separate workstation and that the network connections are too slow or too heavily loaded to allow complete synchronisation between each participant's keystrokes. This means that each workstation will contain a copy of some or all the shared document and that each participant's immediate feedback will be generated locally. However, we still want to give the illusion of synchronous working. We want the participants to work on the same document and we want each participants' edits to appear on the other participants screens even if slightly delayed.

However, the delays in network communications allow race conditions to arise. Figures 5 and 6 show an example of such a race condition. In general the problem is as follows.

We consider two sites A and B and two actions a and b which happen simultaneously, in the sense that network delays mean that neither site knows about the other action when its own action is performed. After some delay each site learns about the others action and must perform some corrective action of its own. First of all each site must decide which action is to be considered as happening 'first'. This may be by use of a global clock or more complex version numbers — the method may differ but it must ensure that both sites agree!

Let us assume that action a is considered to have happened first. Both sites now have a problem. At site A it has an action b which was framed in the context of the state before action a, it must now translate this into some new action $b\prime$ which has the 'same' meaning as the original action. Site A can then simply perform the translated action on its own state. Site B has even more problems as it must in some sense pretend that a happened first. Again it is crucial that both sites come to the same answer!

Similar problems occur in the domain of database transactions. However, such transactions are usually framed in a context independent fashion (although that is debateable) and thus the translation step is not needed. Furthermore if the two actions commute (in the sense that a followed by b is the same as b followed by a) then each site can simply perform the other site's action after its own. In the case when the actions do not commute database systems simply fail one transaction or other and rollback the relevant processes state. Such a solution is clearly not acceptable in a collaborative editing situation. Imagine, you have just typed a few words and then suddenly a warning box appears and your text disappears as your workstation rolls back.

although advocated by some authors [?].

We assume that we start with the situation as it was at the end of Figure 3.

$$display_2 = \boxed{\texttt{Adonis␣is␣}\uparrow\texttt{␣and␣Beatrice␣is␣beautiful}\downarrow\texttt{.}}$$

Beatrice has hit the undo button, we therefore want to work out the actions as they could have been performed. That is we have:

$$\begin{array}{rcl} a & = & op_2 \\ b & = & op_1 \end{array}$$

and we want to calculate:

$$\begin{array}{rcccll} a_b & = & pull_b^{-1}(op_2) & = & op_3 & \text{say} \\ b_a & = & pull_{a_b}(op_1) & = & op_4 & \end{array}$$

First of all we work out the inverse for $pull_1$

$$pull_1^{-1}(pt) = \begin{cases} pt & \text{if } pt < 27 \\ pt + 9 & \text{otherwise.} \end{cases}$$

and obtain op_3:

$$op_3 = pull_1^{-1}(op_2) = insert(\text{``}\texttt{adorable}\text{'}, 10)$$

The operation is in fact unchanged as $pull_1^{-1}(10) = 10$. Thus $op_3 = op_1$ and so $pull_3 = pull_1$. We can now calculate a_b:

$$\begin{array}{rcccl} op_4 & = & pull_3(op_1) & = & insert(\text{``}\texttt{beautiful}\text{'}, pull_3(27)) \\ & & & = & insert(\text{``}\texttt{beautiful}\text{'}, 35) \end{array}$$

This time there is a change op_4 is not the same as op_1. A quick check shows that the effect of op_3 followed by op_4 is indeed the same as that of op_1 folloed by op_2.
We must now work out the inverse action to op_4 in order to undo it. This turns out to be:

$$op_{undo} = delete(9, 35)$$

where $delete(n, p)$ deletes n characters starting at position p. This has the pull function:

$$pull_{undo}(pt) = \begin{cases} pt & \text{if } pt < 35 \\ 35 & \text{if } pt \in [35, 44] \\ pt - 9 & \text{otherwise.} \end{cases}$$

Applying this operation and its pull function to the state at the end of Figure 3, we get:

$$\begin{array}{rcccl} T_{undo} & = & op_{undo}(T_2) & = & \text{`}\texttt{Adonis␣is␣adorable␣and␣Beatrice␣is␣.}\text{'} \\ ap_{undo} & = & pull_{undo}(18) & = & 18 \\ bp_{undo} & = & pull_{undo}(44) & = & 35 \\ display_{undo} & = & \multicolumn{3}{l}{\boxed{\texttt{Adonis␣is␣adorable}\uparrow\texttt{␣and␣Beatrice␣is␣}\downarrow\texttt{.}}} \end{array}$$

Again, this is happily just what we want!

Figure 4: Multi-party undo with dynamic pointers

Adonis and Beatrice are now working on separate machines. They start with their machines showing the same text:

A_0 = `Adonis␣is␣↑␣and␣Beatrice␣is␣.`
B_0 = `Adonis␣is␣␣and␣Beatrice␣is␣↓.`

Simultaneously both users enter their text: Adonis types 'adorable' and Beatrice types 'beautiful'. We label their actions a and b:

$$a = \mathit{insert}(\text{`}\texttt{adorable}\text{'}, 10)$$
$$b = \mathit{insert}(\text{`}\texttt{beautiful}\text{'}, 27)$$

These are processed at their local workstations and each of them receives feedback of their own actions:

A_1 = `Adonis␣is␣adorable↑␣and␣Beatrice␣is␣.`
B_1 = `Adonis␣is␣␣and␣Beatrice␣is␣beautiful↓.`

However, their respective actions are also being transmitted to one another over the network. When they are both received, Adonis' machine processes the action b and Beatrice's machine processes a, leading to:

A_2 = `Adonis␣is␣adorable↑␣and␣Beatbeautifulrice␣is␣.`
B_2 = `Adonis␣is␣adorable␣and␣Beatrice␣is␣beautiful↓.`

Figure 5: Decentralised editing – race conditions

$$\begin{tikzcd}
\text{Adonis' machine} & \text{Beatrice's machine} \\
A_0 \arrow[r, phantom, "="] \arrow[d, "a"'] \arrow[ddr, dashed] & B_0 \arrow[d, "b"] \arrow[ddl, dashed] \\
A_1 \arrow[d, "b"'] & B_1 \arrow[d, "a"] \\
A_2 \arrow[r, phantom, "\neq"] & B_2
\end{tikzcd}$$

Figure 6: Race conditions – diagram

The only published algorithm of which the author is aware, which deals adequately with this problem is Ellis and Gibbs' algorithm as used in the Grove collaborative editing system [5]. This algorithm is described below. However, from the obvious similarity with the undo situation one can see that dynamic pointers can address the required translation process. This is described, but in addition we find that Ellis and Gibbs' algorithm is very nearly an instance of the general algorithm one gets from applying dynamic pointers. Furthermore, on comparison we will find that the pure dynamic pointer solution requires far less effort on the part of the programer and is likely to be more internally consistent.

For the purposes of the following discussion, we will concentrate on the translation at site A, as described above. In theory, the problem for site B can be solved by rolling back to the state before b was performed and then perfoming a followed by $b\prime$. If only the final result of these actions was displayed such an approach would appear reasonable to the users. However, this would be computationally expensive and so in practice one would perform a similar translation of a (say $a\prime$) in order to generate an appropriate action to perform after b. This second translation must satisfy the obvious property that

$$a\ b\prime = b\ a\prime$$

. In practice the two translation processes are nearly identical.

Ellis and Gibs' algorithm

Ellis and Gibbs (1989) recognise and describe the problem in largely similar terms to the description above. Their solution is to look at each possible pair of commands a and b and then give the translated command $b\prime$. They consider only single character commands in their paper as multi-character commands can be built up by repeated application of their rules. For example, their rule for a deletion (b) to be translated through an insertion (a) is as follows:

$$\begin{array}{rcll} a & = & ins(c,n) & \text{— insert character } c \text{ at position } n \\ b & = & del(m) & \text{— delete character before position } m \\ b\prime & = & \begin{cases} del(m) & \text{if } m \leq n \\ del(m+1) & \text{if } m > n \end{cases} & \end{array}$$

Note especially the behaviour when the delettion and insertion are at the same location. The modified deletion will delete the character immediately before the inserted character, not the inserted character itself. If you imagine this as two participants acting at two locations this is the sensible action. The complete set of rules is given in Figure 7.

In addition to these translation rules Ellis and Gibbs describe a method to decide on action ordering using vector timestamps, a technique from distributed systems theory.

Dynamic pointer description

Notice that the rules for Ellis and Gibbs algorithm look very similar to the pull functions described earlier. Indeed, we can give a dynamic pointer solution to the problem. We want to work out what the effect of a command b which was formulated in the context prior to executing a. Now the pull function tells us how pointers preserve their meaning through changing context, so, if we assume that the change of context is captured by the change of locations, then we can simply apply the pull function to all pointers in b:

$$b\prime = pull_a(b)$$

For example, if we consider the single character insert operation ($ins(c,n)$), its pull function is:

$$pull_{ins}(pt) = \begin{cases} pt & \text{if } pt \leq n \\ pt+1 & \text{if } pt > n \end{cases}$$

Ellis and Gibbs consider two primitive commands:

$ins(c,n)$	—	insert character c at position n
$del(m)$	—	delete character before position m

If we are interested in the translation of a more complicated commands say where a is composed of two primitive commands a_1 follwoed by a_2, we first translate b through a_1 and then take the translated command b' and translate that through a_2.
They consider each pair of commands in turn and give a rule for the translated command $b\prime$:

Insert – insert

$$\begin{aligned} a &= ins(c,n) \\ b &= ins(d,m) \\ b\prime &= \begin{cases} ins(d,m) & \text{if } m \le n \\ ins(d,m+1) & \text{if } m > n \end{cases} \end{aligned}$$

Insert – delete

$$\begin{aligned} a &= ins(c,n) \\ b &= del(m) \\ b\prime &= \begin{cases} del(m) & \text{if } m \le n \\ del(m+1) & \text{if } m > n \end{cases} \end{aligned}$$

Delete – insert

$$\begin{aligned} a &= del(n) \\ b &= ins(c,m) \\ b\prime &= \begin{cases} ins(c,m) & \text{if } m < n \\ ins(c,m-1) & \text{if } m \ge n \end{cases} \end{aligned}$$

Delete – delete

$$\begin{aligned} a &= del(n) \\ b &= del(m) \\ b\prime &= \begin{cases} del(m) & \text{if } m < n \\ del(m-1) & \text{if } m > n \\ \text{do nothing} & \text{if } m = n \end{cases} \end{aligned}$$

Note that the notation is different from that in Ellis and Gibbs' paper. Their locations refer to character locations rather than gaps between characters. They also give the corresponding rules for the second translation (which are the same except for the case of two coincident inserts). The above rules have been reformulated using the notation used in this paper and with the correction of a minor error.

Figure 7: Ellis and Gibbs' rules for distributed editing

Note that this is the leave 'em behind version $pull_{-}$.

We can now apply the alogorithm to give the same rule as the Ellis and Gibbs algorithm gave for insert and delete.

$$\begin{array}{rcl} a & = & ins(c, n) \\ b & = & del(m) \\ b\prime & = & del(pull_{ins}(m)) \quad = \quad \begin{cases} del(m) & \text{if } m \leq n \\ del(m+1) & \text{if } m > n \end{cases} \end{array}$$

If we do this for each pair of operations we get almost exactly the same set of rules as Ellis and Gibbs. The only exceptions to this are the cases of two deletes at the same location or two insertions of the same character at the same location. For each of these cases, the Ellis and Gibbs' rules say that the second operation does nothing, whereas the rules based on dynamic pointers would delete or insert a second character. For two deletes the dynamic pointers rule is:

$$\begin{array}{rcl} a & = & del(n) \\ b & = & del(m) \\ b\prime & = & del(pull_{del}(m)) \quad = \quad \begin{cases} del(m) & \text{if } m < n \\ del(m-1) & \text{if } m \geq n \end{cases} \end{array}$$

Imagine the case when the text is '`abcd`', and both participants' cursors are positioned just after the '`c`'. The operations are thus both $del(3)$. Following Ellis and Gibbs rules, one of these would translate to no action giving the result as '`abd`'. The use of dynamic pointers would translate the second delete into $del(2)$, and hence the result would be '`ad`'. We discuss these two options more fully below.

Of course, the dynamic pointers based solution is not limited to single character insert and delete. One can simply follow the same procedure for multicharacter insertions and deletions and for more complex block operations. However, once one begins to deal with block pointers there are situations where the corresponding pull function is not well defined (or at least it is unclear what the correct function is). These situations can be identified in advance and correspond to intersecting block operations. For example, if two participants have overlapping selections and then simultaneously each tries to move their selection somewhere else, what happens to the overlapped text? This suggests that either block selection should induce a temporary lock on the selected text or that all block operations are subject to global syncronisation. Given that these are 'major' operations the slight additional delays (aided by standard soft locking protocols) should not be noticable.

Correct behaviour for coincident deletes and inserts

We saw that the only times that Ellis and Gibbs' algorithm and the use of dynamic pointers disagreed was when there were two deletes or inserts with identical arguments. There are arguments for each position.

Let's take the case of insertion. For example, imagine that the text read '`inteligent`' and each participant has their cursor after the '`l`'. Now if each participant types an '`l`', the dynamic pointer algorithm would generate two additional '`l`'s whereas Ellis and Gibbs' algorithm would generate only one. In this context Ellis and Gibbs are clearly giving the 'right' answer in that the spelling has been corrected.

In contrast, imagine the case when the participants are keeping tallies. Say they each have a pile of questionnaires where the answers are coded A, B or C. The text consists of three lines, where the participants are keeping tallies of the answers. The first line reads '`A:1111`', that is four A answers have been noted so far. Both participants' cursors are at the end of the first line, each reads an A answer in their questionaire and hence hit their '`1`' keys. In this case, the dynamic pointers algorithm is 'right' in that it inserts two characters. It is right in this case because it preserves the invariant that the change in size of the text is the number of inserts minus the number of deletes.

In a sense, the Ellis and Gibbs' algorithm is trying to be 'intelligent', whereas the dynamic pointers algorithm is consistent and simple, but stupid. Of course, the second example above shows that you can never be intelligent enough as what is right depends on the users' intentions.

Finally, it is interesting to consider race conditions at boundaries. Imagine it is Adonis and Beatrice working together and at nearly the same moment they notice the spelling mistake in the word 'inteligent'. At time 0.0 Adonis hits his '`l`' key. At time 1.0 the message arrives at Beatrice's workstation and the insertion is processed. However, simultaneously Beatrice is reaching for her '`l`' key If she hits it at time 1.01, Adonis' insertion will already have been displayed and thus both algorithms will generate two additional characters. If instead she hits it at time 0.99, her insertion will be processed first and then Adonis' insertion will be subject to translation. With Ellis and Gibbs' algorithm this will mean one character whereas with dynamic pointers we get two. Again, whether stability is considered important or whether such race conditions are regarded as a reasonable result of close cooperation is a matter of opinion.

Comparison

Given that both algorithms yield roughly similar results, what advantage is there in adopting the dynamic pointers approach?

generality – dynamic pointers are a more general framework covering several phenomena.

consistency – because the rules are automatically generated they will be more consistent with one another.

extensibility – the dynamic pointer algorithm automatically extends to new operations whereas Ellis and Gibbs require new rules to be written.

cost – if there are n operations, the Ellis and Gibbs algorithm must consider each possible pair, that is there are n^2 rules. However, to apply the dynamic pointers algorithm, one needs to only define a pull function for each operation – n definitions.

To be fair there are counter arguments, the consistency of dynamic pointers may not yield desirable rules. The treatment of coincident deletes is an example. If one decided that the Ellis and Gibbs treatment was better for a particular application then this would have to be added as an exception to the dynaimc pointers scheme. If one considers too many such 'exceptions' then the cost begins to rise.

Perhaps one should not regard the techniques as alternatives. Instead one can think of dynamic pointers as a theoretical underpinning to Ellis and Gibbs' algorithm and a way of generating most of the rules, allowing one to concentrate one's attention on the difficult cases.

Threads

Threads are a way of understanding the distributed editing process. Adonis and Beatrice's machines start off with single version of the shared text. They then do some edits, say Adonis does a_1 and a_2 and Beatrice does b_1 followed by b_2. Until their respective actions are transferred over the network, they temporarily see different versions. However, these versions are lightweight in that they are constantly trying to synchronise with one another and obtain a single shared version. By analogy with lightweight processes, we use the word *thread* for the history of versions which each party sees. The combined interaction can then be seen as a loosely plied yarn, where the threads are sometimes tight together and sometimes slightly separated.

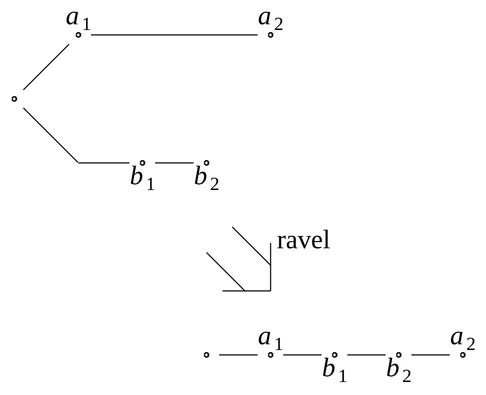

Figure 8: Threads – lightweight versions

Using this analogy we can see the job of the machines when they receive each other's actions as ravelling up the threads which have become separated. Figure 8 shows this process for the four actions. The reason that dynamic pointers help us is that they make this process of ravelling (and unravelling easy). This is because they allow us to interpret actions made in one context in the light of a different, but related context. In fact the diagram is a simplification of the process as it doesn't show that the actions have been suitably modified.

As well as being a way of looking at decentralised activity, threads can help us to understand the undo problem. In the synchronous case, the participants' threads are in a sense ravelled as they go along. In order to perform an undo one must unravel the threads from one another until the required action is found, undo the action and then ravel the threads back up again (see Figure 9).

6 Asynchronous activity

We consider finally the case of asynchronous editing, that is, where several participants may be operating at the same time, but without continuous direct or indirect communication. This might be a prefered form of working, for example, if several authors are working on the same document and wish to work alone and then come together later to compare their work. Alternatively, it may be forced by circumstances, for example, where the participants are working far away with no high speed communications, or where mobile workers are using portable computers and can only synchronise when they return to base. The general problems of such workers are described in more detail elsewhere [3].

The problem

For the rest of this section, we will imagine the scenario where Adonis and Beatrice are writing a book together, but have portable computers which they periodically connect to one another to synchronise their work.

In such circumstances it is possible to avoid problems of concurrent update by ensuring some form of locking or round robin protocol. This would mean that at most one of the authors was updating a particular document (or section) at a time and that the 'control' of the document passed between them only when they synchronised. However, this is often inconvenient, for example, whilst half way through a trans-Atlantic flight, Adonis notices errors in a chapter that Beatrice

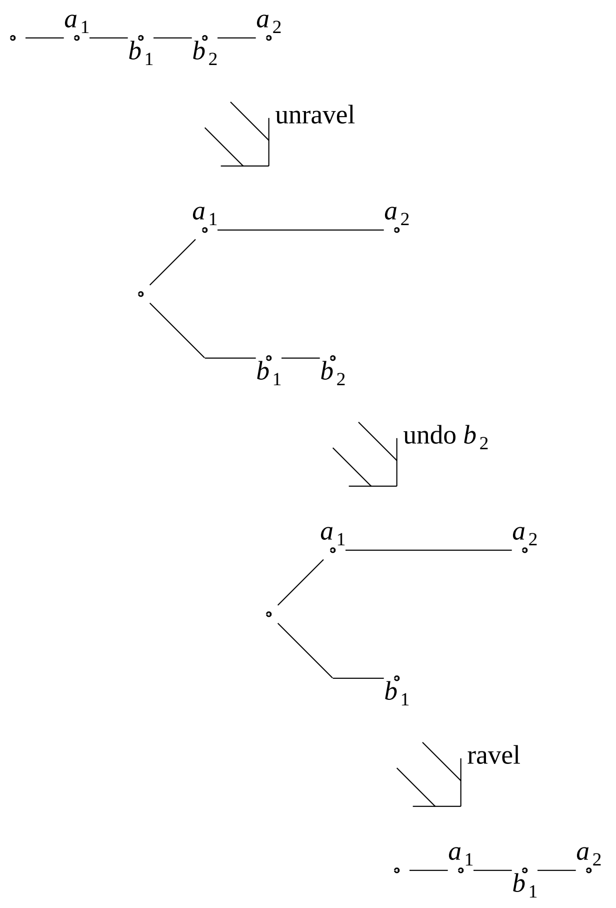

Figure 9: Undo using threads

has a lock on — it may be several days before he can obtain the lock from her!

Alternatively, we might allow either author to edit any chapter at any time. However, sooner or later we will get the situation where on synchronisation, there are two modified versions of the same chapter. How do we resolve the two authors' amendments?

This is rather like the situation with synchronous distributed editing, except the time frame here is much longer, rather than network delays, we are dealing with delays of hours or days. The major difference this makes is in the control over the resynchronisation process. In the synchronous case, we were looking for ways of automatically re-synchronising the concurrent edits, which led to the development of translation algorithms. In the asynchronous case it is often neither possible nor desirable to perform this entirely automatically.

First of all, it may not be possible. Recall that when one considers more complex block oriented operations it may not be possible to perform appropriate transformations. Whereas these awkward cases might arise only infrequently when the delays and therefore the number of overlapping commands are small, in the asynchronous case the size of update is likely to be large and thus the awkard cases become the norm.

Secondly, it is not necessarily desirable to have automatic resynchronisation. Two amendments to different sections which are each sensible in relation to the original contents may not be consis-

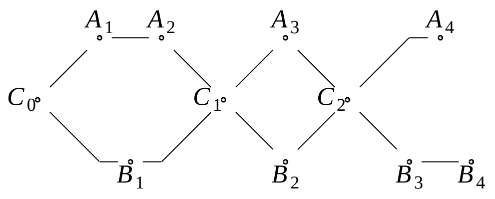

Figure 10: Threads during asynchronous editing

tent with one another. For example, Adonis and Beatrice may both notice that the notation used in sections A and B of their book ar inconsistent. So Adonis edits section A to make it consistent with section B's notation, whilst at the same time Beatrice edits section B to bring into line with the original section A. If the document were automatically merged, no computational difficulties would emerge, but the resulting document would please neither author.

These requirements suggest that automated *support* for the authors is required, but not total automation.

Threads and version control systems

Threads offer a way of talking about this problem. Just as in the case of synchornous distributed editing, we can imagine the thread of versions seen by each author. These threads meet when the authors synchronise their work. In Figure 10, we see Adonis and Beatrice's threads of work. they began with a common document C_0. Then both Adonis and Beatrice worked separately on the document. Adonis edited it to get his version A_1 and then amended this to get A_2. At the same time Beatrice edited the original to get her version B_1. They then got together and synchronised their documents ariving at the common version C_1. This process continued and the current situation is that both have worked since last synchronising (at C_2) and have different current versions. It is time again to resynchronise!

There are thus three threads in Figure10, namely the versions seen by Adonis, those seen by Beatrice and the thread of common synchronised versions:

Adonis' thread	–	$C_0, A_1, A_2, C_1, A_3, C_2, A_4$
Beatrice's thread	–	$C_0, B_1, C_1, B_2, C_2, B_3, B_4$
common thread	–	C_0, C_1, C_2

The presence of the common thread depends on the mode of communication. If communications (when they occur) are not synchronous, there may be no common thread even though the participants threads intertwine in a more complex manner [3]. These more complex cases make a good conceptual and formal understanding even more important, but for the present discussion only the simpler case is considered.

One way of dealing with the problems of resynchronisation is to make the threads explicit to the authors, by using an extend form of version control system [4]. At the point shown in the figure, Adonis' version control system would contain all of his and Beatrice's versions except B_3 and B_4 as these were produced since they last met. Similarly Beatrice's system would know about everything except A_4. When they met, the synchronisation process would take place in two stages. First of all the two version control systems merge, giving rise to the full version graph shown if Figure 10. This merge is totally automatic as the graphs can simply be overlaid. Then, using the information in the joint version control graph, the authors can (with support) merge their latest versions (A_4 and B_4) to give a new common version (C_3).

Merging using dynamic pointers

The merging process is of course quite complex. Elsewhere it has been argued that this must be a 3-way merge [4]. That is, the authors will not only want to look at the two most up-to-date versions, but also the last common point. As they browse through one of these versions, they will want to be able to refer to the equivalent points in the other versions. This is, naturally, a job for dynamic pointers. Indeed, the deltas which are commonly used to store differences in version control systems can give us the requisite pull functions for free.

Consider the task of merging versions A_3 and B_2 to give the new common version C_2. The last common version before A_3 and B_2 is C_1, and the updates to C_1 which gave rise to A_3 and B_2 are a and b respectively. Imagine now, that Beatrice is looking at a partcular point pt in B_2. The system can show her the corresponding point in C_1 by using the inverse pull function $pull_a^{-1}(pt)$. Similarly, it can show her the corresponding point in Adonis' version, A_3, by going forward again $pull_b(pull_a^{-1}(pt))$. As well as allowing browsing, this form of pointer chasing can be used to relate annotations and other forms of locatoon information (such as hypertext links) between versions, even before they are merged.

Translation and conflict

We initally remarked that the problem of asynchronous update was similar to that of synchronous distributed editing, except that totally automatic synchronisation was not desirable. However, the translation process described in Section 5 can be used to aid the user controlled translation process. Consider again the merging of A_3 and B_2. Where some part of a does not conflict with any part of b (which can be detected automatically), the translated operation can be presented to the user as a suggested option. For example, if Beatrice has inserted some text in a section which Adonis has not edited, the suggestion would be that the inserted text was added to the combined document. For the reasons decribed earlier the author doing the merging might not wish to do this, but it is a good default.

In the case where the translation process fails because of conflict (or succeeds, but is questionable), these are areas to focus the authors' attention on during the merge. Basically, the authors would have to explicitly deal with each conflict whereas they could choose to let the system perform the default action on each viable translation.

7 Implementation costs

Dynamic pointers are justifiable purely as a descriptive mechanism. As noted previously virtually any interactive system will have features which are best described as a form of dynamic pointer. Such features can only be improved by using dynamic pointers as part of the design process, but they need not be explicit in the implementation. However, in addition to this formal descriptive rôle, it is possible to use dynamic pointers as an explicit implementation structure. The advantages of this are:

- If dynamic pointers are in the implementation, then many of the advantages of consistency obtained by using them as design method arise naturally and without further effort.
- They make it easy to implement algorithms such as Ellis and Gibbs scheme.
- They are a natural part of the programmer's interface to group widgets (gwidgets) for editing. (Note that text editing widgets are notoriously inflexible, especially for groupware purposes.)

Of course, the adoption of dynamic pointers as an implementation mechanism has associated costs as one must find ways of efficiently representing them. In particular, the way dynamic pointer like features are often coded is as some implicit part of the data structure coding the object of interest. Separarting the object from the pointers will have obvious performance costs. However, it is just this over-intimate connection which makes it difficult to modify single user applications and widgets for multi-user use.

There are two ways to handle dynamic pointers in an implementation:

1. Explicitly represent the pull function, which largely presupposes that the different versions have some explicit representation in the implementation.

2. Implicitly code pointers by having a number of 'currencies' associated with an object which it updates accordingly as the object is modified.

The latter option allows more efficient implementation of pointers, but can only handle known pointers and is inflexible. The former approach is thus essential when wanting to use new pointers to refer to their corresponding location in old versions of an object. For groupware, option 2 is only really viable in synchronous editing. However, there is no reason why both forms should not be available especially when a heavily resuable gwidget is being designed.

One major problem with implementing dynamic pointers is that the time taken to perform an update can be proportional to N, the total number of pointers in the system. This occurs as one has to individually operate on each pointer with the pull function. However, with care one can ensure times proportional to n, the number of pointers within the area changed by the update.

Although the author has so far only used dynamic pointers in the analysis of groupware, he has experience of their use in the implementation of single user interfaces. A large case study made particularly strong use of dynamic pointers (indeed part of its purpose was to test their utility). A multi-windo editor was designed using a three layered approach: unformated text, formated text and screen display. This implementation made use of several thousand dynamic pointers at any moment and used crude codings of the pointers. Despite this it achieved reasonable real-time response. In groupware applications where the gains are higher dynamic pointers are thus a good candidate as an implementation mechanism.

8 Summary

We have seen that dynamic pointers give us a uniform way of analysing: synchronous editing, multi-user undo, distributed and asynchronous editing. In the case of synchronous distributed editing we saw how Ellis and Gibbs' algorithm was closely related to the algorithm one gets almost for free when using dynamic pointers to represent operations. Dynamic pointers are particularly good at representing multiple viewpoints and reasoning about mappings between different versions of the same object. This is useful for single-user applications but essential for understanding groupware. So far dynamic pointers have only been used explicitly in the implementation of single user applications, but the indications are that they will be especially important in building reusable groupware widgets (gwidgets).

In addition, threads are a useful way of looking at situations where parallel versions develop because of concurrent activity. As with dynamic pointers, threads are useful not only as a descriptive mechanism, but can be used (in the form of extended version control graphs) as part of the implementation of certain classes of groupware.

In short, dynamic pointers and threads are extremely important as specification and descriptive mechanisms for groupware and also have a string part to play in its implementation.

References

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

[2] A.J. Dix. *Formal methods for interactive systems.* Academic Press, 1991.

[3] Alan Dix and Russell Beale. Information requirements of distributed workers. In A. Dix, editor, *Remote cooperation — CSCW issues for mobile and tele-workers.* Springer Verlag (to appear), 1993.

[4] Alan J. Dix and Victoria C. Miles. Version control for asynchronous group work. Technical Report YCS 181, Computer Science Dept., University of York, U.K., 1992. (Poster presentation HCI'92: People and Computers VII).

[5] C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. *SIGMOD Record,* 18(2):399–407, June 1989. 1989 ACM SIGMOD International Conference on Management of Data.