Editing Text Versions With Layers

Marius Nita
marius@cs.washington.edu

Draft, August 12, 2010

Abstract

Layers is a nascent text editing concept that combines text versioning into the editing process. Its versioning model is always-merged, conflict-free, and fully visible and manipulable within the editing interface. This report describes the underlying model and some of its properties, the implementation, the user interface, and applications to software engineering.

1 Introduction

Maintaining multiple alternative versions of a text is a fundamental aspect of the authoring process in text-centered disciplines like programming, copywriting, and editing. Some scenarios are:

- A client of a Web firm may request multiple alternative implementations of a concept before deciding on any implementation. Then, copywriters produce multiple versions of the copy, layout engineers produce multiple versions of layout templates, and programmers produce multiple versions of their code. When the client has decided on one version, the extraneous versions are discarded from mainline production.

- When implementing a new feature, a programmer may pursue multiple implementation strategies before deciding on any one of them. New knowledge gained in the act of programming can render the current implementation strategy infeasible or undesirable, driving the programmer to either revert to an old version, start over, or split the current implementation path into two subpaths. After comparing and contrasting the alternatives, the programmer chooses the best one and discards the others.

- A text artifact usually has *development versions*, versions that are only relevant in the context of authoring. Programs, for example, have debugging and logging modes that are excluded in the rendering of the production versions. Other text documents will contain notes, self-reminders, and other meta-text that is discarded in the final output.

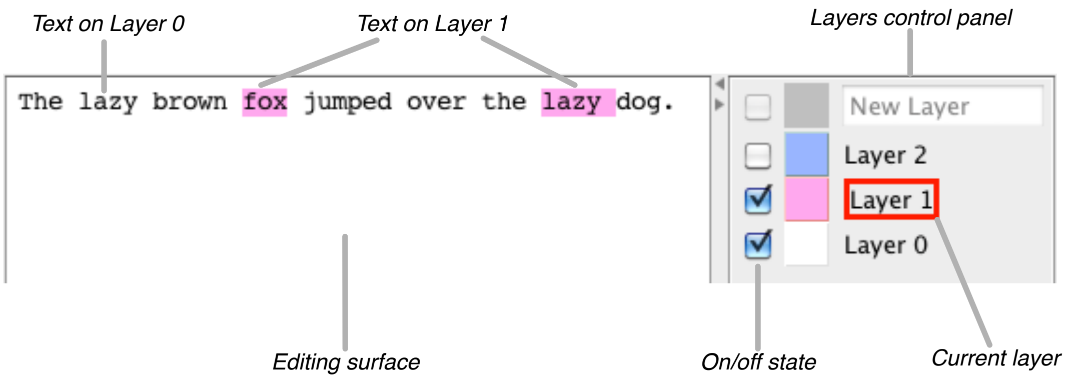
A crude method for managing multiversion texts is to copy a document multiple times and perform changes on the copies. A `diff`-like tool can be used for post-hoc identification and explanation of the changes. However, managing multiple copies gives rise to an exponential explosion of versions. For example, if 3 changes are made to separate parts of the document, up to 8 (2^3) copies are to be maintained. Moreover, if a hard change is made (a change that does not define a version, but rather modifies the document in place) to a part of the text that is common to all the versions, then all the versions will have to be changed consistently. If one desires to understand how one version simultaneously relates to the other versions, a n-way diffing tool is required. With these complications, authors can quickly lose knowledge pertaining to the locations and effects of their changes.

Most programmers are familiar with version management tools like SVN, Git, Mercurial, CVS, etc. These tools employ the concept of *branches*, which can be used to maintain an unlimited number of related versions that can be edited independently. Branches are logically separated, however. While working in one branch, it is not apparent how the edits relate to and affect the other branches. Moreover, if an edit is made to the common parts of two related branches, the edit remains isolated to one branch and has to be manually migrated to the other. Branching therefore fails to provide the author with a generally coherent picture for where the edits are located and how the edits relate the possibly many branches. Branching is useful for performing changes to a document in isolation, ignorant of changes in other branches, but is unnatural for working from a birds-eye view, where the effects of editing in one version are clearly understood in the context of the other versions. Currently, branching is most widely used to temporarily manage interference among authors working on the same project, and much less to express the versioned structure of documents.

These approaches do not help authors comprehend the impact of their edits on the version structure, at the time of editing. The unknown version structure, lurking in the background, may decide the edits to be invalid only at a later time, upon realization of a version conflict. Therefore, authors are generally in a noncommittal state of mind, accepting their edits as possibly wrong or false in the future.

We call an editing system *version-coherent* when its user knows the impact of edits in real-time, without the chance of being invalidated by a future conflict. In short, version coherence combines the comfort of single-version editing into a multi-version structure: every edit is final and will not be explicitly or implicitly challenged by the system. In versioning systems, there is a fundamental tension between version coherence and multiple concurrent authors: multiple authors may want to edit in isolation, paying the price of post-hoc manual conflict resolution.

This paper describes layers, a nascent text-editing concept that seamlessly combines content authoring with lightweight version management. With layers, versions are *always merged* into a central structure that is updated with every keystroke. Because they are always merged, and therefore never in conflict, layered documents are naturally version-coherent. When using layers, the user modifies each version directly by editing, and controls the versioning structure via a simple auxiliary interface. For the time being, layers is aimed at single users: there is no explicit consideration for several authors editing separate versions



The lazy brown fox jumped over the lazy dog.

Figure 1: Screenshot of a simple prototype implementation of layers.

at the same time. There may be a sweet spot or a configurable sliding scale between version coherence and isolated editing. This is left for future work.

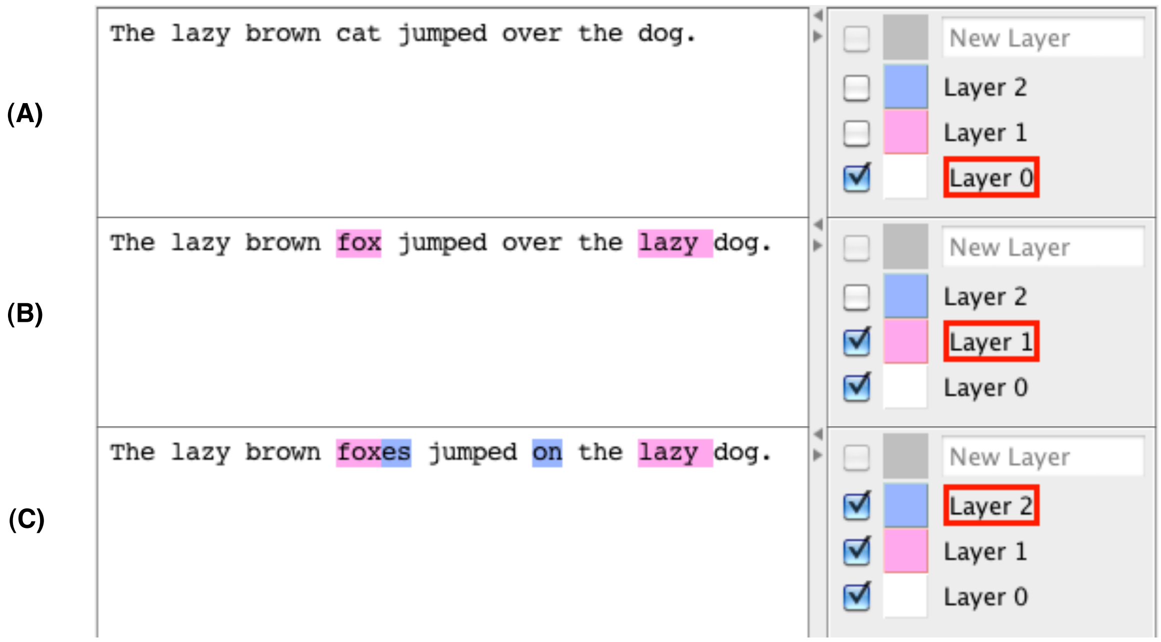
The rest of the paper is organized as follows. Section 2 gives a high-level overview of the layers concept. Section 3 describes the TCM, the theoretical model underlying layers. Section 4 describes an efficient implementation strategy for the TCM. Section 5 considers user interface features and enhancements. Section 6 discusses applications for layers in the software engineering space. Section 7 discusses related work, and Section 8 concludes. We use the sans-serif word layers to denote the concept as distinguished from individual layers.

2 Concept Overview

We will describe layers in the context of a research prototype implementation that embodies key aspects of the concept. A screenshot of the prototype is shown in Figure 1. On the left side of the screenshot, there is a traditional editing surface, like we would find in any text editor. Text can be inserted and deleted using the usual keyboard and mouse actions. On the right is the layers control panel, which allows layers to be created, named, turned on, turned off, deleted, and assigned a color.

A *layer* is a named set of edits. An edit is an insertion or a deletion of text, performed upon the editing surface in the usual way. Sometimes, an insertion and a deletion together are conceptualized as a replacement, but replacement needn't be a separate entity. In Figure 1, the two edits belonging to Layer 1 are highlighted in pink, the layer's corresponding color. The rest of the text is highlighted in white and thus belongs to Layer 0. In the control panel, we see that Layers 0 and 1 are turned on, and Layer 2 is turned off. When a layer is turned off, its corresponding edits disappear from the editing surface. Essentially, the state of the editing surface is as if Layer 2 had never been created.[1]

[1]We will see later that it's possible and desirable to include information about turned-off layers in the editing surface.



```
The lazy brown cat jumped over the dog.
```

```
The lazy brown fox jumped over the lazy dog.
```

```
The lazy brown foxes jumped on the lazy dog.
```

Figure 2: A sample document in the layers prototype. (A) Only Layer 0 is turned on. (B) Layers 0 and 1 are turned on. (C) All three layers are turned on.

The red box in the control panel area denotes the *current layer*. Any edits performed upon the editing surface belong to the current layer and are immediately highlighted in the layer's color. Therefore, the current layer is necessarily turned on. Finally, when clicking in the box labeled "New Layer," the box becomes an active, empty text field. Typing a name and pressing Enter creates a new layer with that name. By clicking the color box, a new color can be assigned to a layer. Its corresponding edits in the editing surface are recolored accordingly.

Figure 2 shows the document in Figure 1, reconfigured three times to form three alternative versions of the text, using layers. No editing was performed between the three screenshots; the different document versions are the results of combining different layers using the on/off buttons in the control panel to the right of the editor. In Figure 2(A), only Layer 0 is turned on. We therefore see only the edits belonging to Layer 0. In Figure 2(B), Layers 0 and 1 are turned on, so we see their edits combined. Conceptually, there are two edits on Layer 1. The first deletes the word `'cat'` and inserts `'fox'` in its place. The second inserts `'lazy '` before `'dog'`. In Figure 2(C), all three layers are turned on. The third layer inserts the text `'es'`, deletes the word `'over'`, and inserts `'on'` in its place.

An important point to emphasize here is that layers are *not* versions. Versions are instead created by *combining* layers, and one layer may participate in many versions. (In our example, Layer 0 participates in all three versions and Layer 1 participates in two of the versions.) In general, there are $2^{\#\mathrm{Layers}}$ versions, including the null version in which all layers are off. At any time, the contents of the editing surface form one version of the text. To transition the document to another version, the user changes the on/off layer state in the control panel. The layers prototype has two types of "Save" commands. One saves to disk the current version (the contents of the editing surface) into a flat text file. The other saves the underlying document representation.

2.1 Further Design Considerations

The layers concept, as presented so far, raises a few more questions:

1. *Implicit dependencies:* Because Layer 1 removes the text `'cat'` inserted by Layer 0, and supposing that Layer 0 is off, does it make sense for Layer 1 to be on? In some situations it may be desirable to link layers in this way. A layer may not make sense in isolation, so having to manually identify and turn on the layers upon which it logically depends is unnecessarily laborious. In other situations, the user may wish to view and edit the layer in isolation. For example, a programmer may want to know the functions that are called on a layer, regardless of where. Text deleted by the layer may be shown marked as deleted, without the context where the deletion occurs.

2. *Dependency structure:* On the question of representation, are layers organized in a hierarchy, implicitly dependent upon one another, or are they in a flat, set-like, relative relationship? It seems that because edits in one layers generally depend on edits in other layers, they should be in a hierarchy. However, as discussed above, from the perspective of *uses* it seems that regardless their logical relationship, it is often useful to treat layers as flat: each independently viewable, without the system imposing by rule that one layer can only be viewed in the context of another.

3. *Multiple layers deleting the same text:* If Layer 1, which deletes the word `'cat'` from Layer 0, is off, and Layer 2 is on, does it make sense for Layer 2 to delete (part of) the text `'cat'`? In version control systems, this a key part of the definition of *conflict*: if two branches edit the same line, they are in conflict. Because layers works at the granularity of characters, the only way for two layers to come into conflict is to delete the same character. The good news is that this doesn't matter! Deleting a character twice is as good as deleting it once or n times. Therefore, layers has no notion of conflict and merging. A layered document is always merged and therefore always conflict-free.

4. *Positional ambiguities:* If Layer 1 is off and Layer 2 is on, and Layer 2 inserts the text `'red '` at the same position at which (the currently invisible) `'lazy '` is inserted by Layer 1, should `'red '` appear before or after `'lazy '` when both layers are on? More generally other words, editing a document when other edits are not visible can introduce ambiguities. In programming, if calls to a logging library are kept on a layer, but the majority of the time the logging layer is off, it seems sensible that the programmer should have some indication of *where* the calls are, and perhaps be given the choice to insert code before or after the invisible call, without having to turn on the layer.

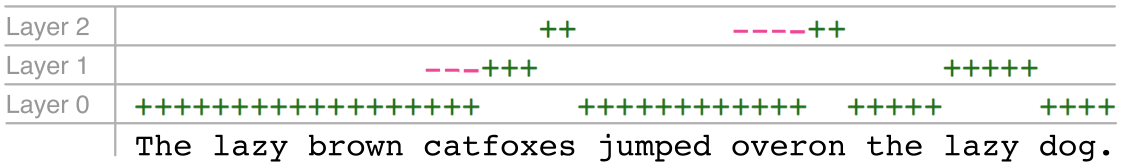


Figure 3: Underlying document structure for the example in Figure 2.

Leveraging the experience gained with our prototype, we began developing the Tagged Character Model, a core calculus for layers that allows formal reasoning about system invariants, experimentation in the design space sketched in this section, and a path to an efficient implementation.

3 Tagged Character Model

Figure 3 shows a pictorial view of one way to conceptualize the representation of the layered document shown in Figures 1 and 2. The document is a sequence of characters, where each character is marked above by the layer that inserted it (with a `+` sign) and by the layers that deleted it (with a `-` sign). Intuitively, to render a document on the editing surface, we traverse the sequence of text and for each character, we combine the `+`s and `-`s in its column. If there is a `-` in the column, or neither `+`s nor `-`s, the character is rendered invisible. The `+`s and `-`s belonging to layers that are turned off are ignored in the rendering.

The Tagged Character Model (TCM) is a lightweight calculus that represents layered documents in the spirit of the figure above. In the TCM, a layer is nothing but a name, or *tag*, and a document is a sequence of characters tagged by the layer that added it (marked by a `+` sign above) and by the layers that removed it (marked by a `-` sign). The TCM's core document representation is flat, keeping all the edits of all the layers in one flat sequence. The TCM addresses questions 1–2 of Section 2.1 by leaving dependencies out of the representation and considering all layers as independently viewable/editable. User-controlled dependency structure can be added, as discussed in Section 5. It answers question 3 by naturally allowing multiple layers to delete the same text. Question 4 is addressed in Section 5.

3.1 Core definitions

Figure 3.1 shows the syntax of the TCM. A symbol k is either c^T, a character tagged by tag T, or a tagged character that is marked either by hidden or dangling. The hidden mark is used for hiding inserted text when the corresponding layer is off, or hiding deleted text when the corresponding layer is on. The dangling mark is used for denoting deleted text

$$\begin{array}{rcll}
k & ::= & c^T \mid \mathsf{hidden}(c^T) \mid \mathsf{dangling}(c^T) & \text{(symbol)} \\
T & ::= & +t - \bar{t} & \text{(tag)} \\
D & ::= & \epsilon \mid k \cdot D & \text{(document)} \\
e & ::= & \text{insert}(c) \mid \text{delete} & \text{(editing actions)} \\
A & ::= & \text{Turn on } t \mid \text{Turn off } t \mid \text{Switch to } t \mid \text{Edit } e & \text{(system actions)} \\
On & \in & t \rightarrow \{\text{true}, \text{false}\} & \text{(layer state)} \\
S & ::= & D; On; t & \text{(system state)} \\
t & \in & \text{layer names} & \\
c & \in & \text{text characters} &
\end{array}$$

Figure 4: Layers core syntax. $\bar{t}$ denotes a set of t.

when the layer that the text is *deleted from* is off. A tag T is $+t - \bar{t}$, meaning "inserted by layer t and subsequently deleted by layers $\bar{t}$. ($\bar{t}$ is a set of tags t.) A document D is a list defined inductively: either empty (ϵ) or a symbol followed by a document ($k \cdot D$). One core assumption of the TCM, evident in the tag syntax $+t - \bar{t}$, is that it does not allow text to be added by more than one layer. Intuitively, there is only one "current layer." However, this assumption may be relaxed in the future if simultaneous insertion on multiple layers is found useful.

In the TCM, the core representation of the running example from prior sections is as follows (where $+0$ is shorthand for $+0 - \emptyset$, $+0 - 1$ shorthand for $+0 - \{1\}$, and $\Box$ denotes a space):

$$\texttt{T}^{+0}\texttt{h}^{+0}\texttt{e}^{+0}\Box^{+0}\texttt{l}^{+0}\texttt{a}^{+0}\texttt{z}^{+0}\texttt{y}^{+0}\Box^{+0}\texttt{b}^{+0}\texttt{r}^{+0}\texttt{o}^{+0}\texttt{w}^{+0}\texttt{n}^{+0}\Box^{+0}\texttt{c}^{+0-1}\texttt{a}^{+0-1}\texttt{t}^{+0-1}\texttt{f}^{+1}\texttt{o}^{+1}\texttt{x}^{+1}\texttt{e}^{+2}\texttt{s}^{+2}\Box^{+0}$$
$$\texttt{j}^{+0}\texttt{u}^{+0}\texttt{m}^{+0}\texttt{p}^{+0}\texttt{e}^{+0}\texttt{d}^{+0}\Box^{+0}\texttt{o}^{+0-2}\texttt{v}^{+0-2}\texttt{e}^{+0-2}\texttt{r}^{+0-2}\texttt{o}^{+2}\texttt{n}^{+2}\Box^{+0}\texttt{t}^{+0}\texttt{h}^{+0}\texttt{e}^{+0}\Box^{+0}\texttt{l}^{+2}\texttt{a}^{+2}\texttt{z}^{+2}\texttt{y}^{+2}\Box^{+2}$$
$$\texttt{d}^{+0}\texttt{o}^{+0}\texttt{g}^{+0}\texttt{.}^{+0}$$

The symbol $\texttt{c}^{+0-1}$, for example, denotes the fact that the character $\texttt{c}$ was initially inserted by Layer 0 and subsequently deleted by Layer 1. This representation contains all the information in the layered document. Tagged characters marked by hidden or dangling are not present because they are introduced only when layers are turned on and off, as we will see shortly. In this example, layers are neither on nor off.

An editing action e is either insert(c), which inserts character c at the current position, and delete, which deletes the character at the current position.[2] A system action A represents actions that can be taken by the user at the top level. "Turn on t" and "Turn off t" are actions that turn layer t on or off. "Switch to t" makes t the current layer, and "Edit e" performs the editing action e upon the editing surface. The function On decides which layers are on. The state S represents the top-level state of the whole editing system and is a triple of a document D, the state function On, and the current layer t.

Next, I show how to perform edits upon a TCM document:

[2] "Current position" will become more clear shortly.

Definition 1 $\boxed{\mathit{Edit}(e_t, D) = D'}$

$$
\begin{array}{lcll}
\mathit{Edit}(\mathrm{insert}_t(c), D + [k] + D') & = & D + [c^{+t-\emptyset}] + [k] + D' & \textbf{(ins)} \\
\mathit{Edit}(\mathrm{delete}_t, D + [c^{+t'-\bar{t}}] + D') & = & \textsf{if } t = t' \textsf{ then} & \textbf{(del1)} \\
 & & \quad D + D' & \\
 & & \textsf{else} & \\
 & & \quad D + [\textsf{hidden}(c^{+t'-\bar{t}\cup\{t\}})] + D' & \\
\mathit{Edit}(\mathrm{delete}_t, D + [\textsf{hidden}(c^T)] + D') & = & D + [\textsf{hidden}(c^T)] + D' & \textbf{(del2)} \\
\mathit{Edit}(\mathrm{delete}_t, D + [\textsf{dangling}(c^T)] + D') & = & D + [\textsf{dangling}(c^T)] + D' & \textbf{(del3)}
\end{array}
$$

The function $\mathit{Edit}(e_t, D)$ takes an editing action e, current layer t, and a document D and yields the modified document D'. There is no explicit notion of "current position" or "cursor position" in this model, so we implicitly assume that the cursor position is picked nondeterministically, by decomposing the document into $D+[k]+D'$, where k is the symbol under the cursor, D is the portion of the document preceding it, and D' is the portion of the document succeeding it. We take $[k]$ to mean the document containing the single symbol k and $D+D'$ the list concatenation of D and D'.

The rule (ins) simply inserts a fresh symbol $c^{+t-\emptyset}$, regardless of the symbol under the cursor. A more thorough definition could handle "undeletes." For example, if the symbol under the cursor was hidden and layer t was one of the layers hiding it, we could simply delete t from its set of removers. In this model, we opt for theoretical simplicity by allowing documents to be larger than absolutely necessary, but without affecting expressive power.

The rule (del1) deletes the current character. If the current character was inserted by t, the current layer, the character is simply removed. Otherwise, t is added to its set of removers and the symbol is marked as hidden. Intuitively, deleting text belonging to another layer simply hides the text from view. Rules (del2) and (del3) encode the fact that deletes do nothing when performed on hidden or dangling symbols.

Next, I show how the document is *filtered* through a state function *On*, revealing and obscuring symbols according to the state function.

Definition 2 $\boxed{\mathit{Filter}(\mathit{On}, D) = D'}$

$$
\begin{array}{lcl}
\mathit{Filter}(\mathit{On}, \epsilon) & = & \epsilon \\
\mathit{Filter}(\mathit{On}, c^{+t-\bar{t}} \cdot D) & = & \textsf{if } (\mathit{On}(t) \wedge \exists t' \in \bar{t}. \mathit{On}(t')) \vee (\neg \mathit{On}(t) \wedge \forall t' \in \bar{t}. \neg \mathit{On}(t')) \textsf{ then} \\
 & & \quad \textsf{hidden}(c^{+t-\bar{t}}) \cdot \mathit{Filter}(\mathit{On}, D) \\
 & & \textsf{elseif } \mathit{On}(t) \wedge \forall t' \in \bar{t}. \neg \mathit{On}(t') \textsf{ then} \\
 & & \quad c^{+t-\bar{t}} \cdot \mathit{Filter}(\mathit{On}, D) \\
 & & \textsf{elseif } \neg \mathit{On}(t) \wedge \exists t' \in \bar{t}. \mathit{On}(t') \textsf{ then} \\
 & & \quad \textsf{dangling}(c^{+t-\bar{t}}) \cdot \mathit{Filter}(S, D) \\
\mathit{Filter}(\mathit{On}, \_) & = & \textsf{ERROR}
\end{array}
$$

The function *Filter*(On, D) takes a state function *On* and a document D and yields the resulting *filtered document* D'. The filtered document fully retains all of the information in D, except it marks some of the characters as hidden or dangling. An implementation can choose how to display hidden and dangling symbols to the user. In the layers prototype, they are simply not shown.

Filtering an empty document (ϵ) yields an empty document. When filtering a symbol $c^{+t-\bar{t}}$, we mark the symbol as hidden when t (the layer that inserted c) and some layer in $\bar{t}$ are both on, or when t and all the layers in $\bar{t}$ are all off. Intuitively, either the character is shown by an insert and then obscured by a delete on another layer, or it's not shown, because t is off, and is also not obscured, because all the layers in $\bar{t}$ are off. If t is on and all the layers in $\bar{t}$ are off, the symbol is left unmarked. When a symbol is marked dangling, it means that the layer that inserted it is off and some layer that deletes it is on.

Filter rejects documents that contain dangling or hidden symbols, therefore it cannot be applied to an already filtered document. Before a filtered document can be filtered through another *On* function, it has to be unfiltered:

Definition 3 $\boxed{\mathit{Unfilter}(D) = D'}$

$$\begin{array}{lcl}
\mathit{Unfilter}(\epsilon) & = & \epsilon \\
\mathit{Unfilter}(c^{+t-\bar{t}} \cdot D) & = & c^{+t-\bar{t}} \cdot \mathit{Unfilter}(D) \\
\mathit{Unfilter}(\mathsf{hidden}(c^T) \cdot D) & = & c^T \cdot \mathit{Unfilter}(D) \\
\mathit{Unfilter}(\mathsf{dangling}(c^T) \cdot D) & = & c^T \cdot \mathit{Unfilter}(D)
\end{array}$$

The function *Unfilter*(D) undoes the action of *Filter* by removing all dangling and hidden marks, bringing the document back to a state ignorant of whether layers are on or off. Intuitively, filtering followed by unfiltering should leave the document unchanged. This is stated formally as follows:

Theorem 4 (Unfilter undoes Filter) *Supposing D contains no dangling or hidden symbols, then*

1. *Unfilter*$(\mathit{Filter}(On, D)) = D$.

2. *Filter*$(On', \mathit{Unfilter}(\mathit{Filter}(On, D))) = \mathit{Filter}(On', D)$.

Part (2) of the theorem is a direct consequence of (1) and essentially reveals the process by which a filtered document is re-rendered when some layers are turned on and off. Given a new function On' (acquired by turning some layers on or off), the document, previously filtered through On, is first unfiltered and then re-filtered through On'. Part (2) of the theorem shows this to be the same as filtering through On' to begin with.

Finally, I show the top-level editing semantics:

Definition 5 $\boxed{D; On; t \vdash A \rightarrow D'; On'; t'}$

$$\begin{array}{llllr}
D; On; t & \vdash & \text{Turn on } t' & \rightarrow & \mathsf{let}\ On' = Change(On, t' \rightarrow \text{true})\ \mathsf{in} & \textbf{(on)} \\
 & & & & \mathsf{let}\ D' = Filter(On', Unfilter(D)) & \\
 & & & & \mathsf{in}\ D'; On'; t & \\
D; On; t & \vdash & \text{Turn off } t' & \rightarrow & \mathsf{let}\ On' = Change(On, t' \rightarrow \text{false})\ \mathsf{in} & \textbf{(off)} \\
 & & & & \mathsf{let}\ D' = Filter(On', Unfilter(D)) & \\
 & & & & \mathsf{in}\ D'; On'; t & \\
D; On; t & \vdash & \text{Switch to } t' & \rightarrow & D; On; t' & \textbf{(switch)} \\
D; On; t & \vdash & \text{Edit } e & \rightarrow & \mathsf{if}\ \neg On(t)\ \mathsf{then} & \textbf{(edit)} \\
 & & & & \qquad \mathsf{ERROR} & \\
 & & & & \mathsf{else} & \\
 & & & & \qquad Edit(e_t, D); On; t &
\end{array}$$

The relation $D; On; t \vdash A \rightarrow D'; On'; t'$ means that starting in a state $D; On; t$ and performing the action A yields the state $D'; On'; t'$. That is, due to A, the document may have changed to D', the state function may have changed to On', and the current layer may have changed to t'.

Rules (on) and (off) show how layers are turned on and off. The function $Change(On, t \rightarrow \text{true})$ is identical to On except $On(t)$ is now true, if it wasn't already. To turn a layer on, a new state function On' is produced that maps the layer in question to true. The current document D is then unfiltered, and the result filtered back through On'. The rule (off) is the same, except that the layer in question is mapped to false.

The rule (switch) simply changes the system state by replacing the current layer with t'. In this model, switching does not by itself ensure that the switched-to layer is on. Therefore, there exists the possibility of attempting to type on a layer when the layer is off. The last rule, (edit), therefore ensures that the current layer is on before executing the editing action. If the current layer is on, the editing action e is tagged with layer t and the editing action is applied to the document by calling $Edit(e_t, D)$.

3.2 Validity and editing non-interference

We would like to prove an important non-interference property: turning on a layer and performing edits on it leaves the previous *visible* state of the document (when the layer was off) unchanged. First, we need to define validity of system states:

Definition 6 $\boxed{Valid(D; On; t) = \{true, false\}}$

$$\begin{array}{lcl}
Valid(\epsilon; On; t) & = & \text{true} \\
Valid(c^{+t-\bar{t}} \cdot D'; On; t') & = & On(t) \wedge \forall t'' \in \bar{t}.\neg On(t'') \wedge Valid(D'; On; t') \\
Valid(\mathsf{hidden}(c^{+t-\bar{t}}) \cdot D'; On; t') & = & (On(t) \wedge \exists t'' \in \bar{t}.On(t'') \vee \\
 & & \quad \neg On(t) \wedge \forall t'' \in \bar{t}.\neg On(t'')) \wedge Valid(D'; On; t') \\
Valid(\mathsf{dangling}(c^{+t-\bar{t}}) \cdot D'; On; t') & = & \neg On(t) \wedge \exists t'' \in \bar{t}.On(t'') \wedge Valid(D'; On; t')
\end{array}$$

A system state $D; On; t$ is *valid* when the current state of the document D is in full agreement with the On function. An empty document is always valid. A symbol $c^{+t-\bar{t}}$ is valid when t is on and all the layers in $\bar{t}$ are off. A symbol $\mathsf{hidden}(c^{+t-\bar{t}})$ is valid either when t is on and some layer in $\bar{t}$ is on, or when t is off and all layers in $\bar{t}$ are off. A symbol $\mathsf{dangling}(c^{+t-\bar{t}})$ is valid when t is off and some layer in $\bar{t}$ is on.

Essentially, the validity of a state $D; On; t$ is established by assuming D had been the result of filtering a D' through On and recording the conditions that $Filter$ imposes on its output. Naturally, validity has an operational definition in terms of $Filter$ and $Unfilter$:

Theorem 7 (Operational Validity) $Valid(D; On; t)$ *iff* $Filter(On, Unfilter(D)) = D$

Previously, we proved that $Unfilter(Filter(On, D)) = D$ in all cases. Operational validity shows that the reverse, $Filter(On, Unfilter(D)) = D$, is only true when $D; On; t$ is valid. Intuitively, $D; On; t$ is valid only when D has already been filtered through On, so unfiltering and re-filtering through On does nothing.

The following theorem establishes the crucial property that validity is never lost in editing:

Theorem 8 (Editing preserves validity) If $Valid(S_1)$ and $S_1 \vdash A_1, \ldots, A_n \rightarrow S_2$, and $S_2 \neq \mathsf{ERROR}$, then $Valid(S_2)$.

That is, executing any number of system actions, starting in a valid state S_1 and ending in a state S_2, S_2 is also valid, provided it isn't the ERROR state. If validity weren't preserved by editing, then any reasoning about the relationship between D and On in the middle of editing would be generally false.

Before we can state the non-interference theorem, we need a way to establish equivalence among the *visible renderings* of two documents. That is, two documents may be different when compared point-wise, but may appear as identical to the user, their differences being in the hidden text. The following function serves this purpose:

Definition 9 $\boxed{Save(D) = Plaintext}$

$$\begin{array}{lcl} Save(\epsilon) & = & \epsilon \\ Save(c^{+t-\bar{t}} \cdot D) & = & c \cdot Save(D) \\ Save(\mathsf{hidden}(c^T) \cdot D) & = & Save(D) \\ Save(\mathsf{dangling}(c^T) \cdot D) & = & Save(D) \end{array}$$

The function $Save(D)$ discards all tags and ignores all hidden and dangling symbols, saving only the visible text, in the order in which it appears in the document. It is likely that $D \neq D'$ when $Save(D) = Save(D')$. For example, D may contain twice as many hidden symbols as D', but they are all ignored by $Save$.

The non-interference theorem is stated as follows:

Theorem 10 (Editing non-interference) If a layer is initially turned off, and we turn it on, switch to it, perform any number of edits, and turn it off, then the initial rendered (saved) state of the document remains unchanged.

Suppose $Valid(D_1; On_1; t_1)$ and $On_1(t_2) =$ false. Then we take the following steps:

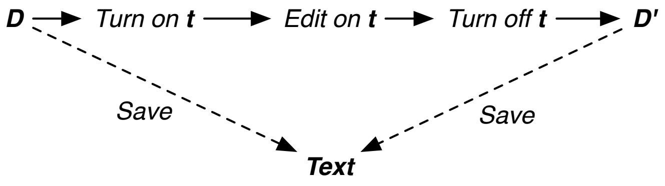


Figure 5: Layers non-interference depicted graphically. Edits on a layer leave the previous state of the document unchanged under *Save*.

1. $D_1; On_1; t_1 \ \vdash$ Turn on $t_2 \rightarrow D_1; On_2; t_1$.
2. $D_1; On_2; t_1 \ \vdash$ Switch to $t_2 \rightarrow D_1; On_2; t_2$.
3. $D_1; On_2; t_2 \ \vdash$ Edit $e_1, \ldots,$ Edit $e_n \rightarrow D_2; On_2; t_2$.

Then, if we perform the following steps,

4. $D_2; On_2; t_2 \ \vdash$ Turn off $t_2 \rightarrow D_3; On_1; t_2$.
5. $D_3; On_1; t_2 \ \vdash$ Switch to $t_1 \rightarrow D_3; On_1; t_1$.

$\mathit{Save}(D_1) = \mathit{Save}(D_3)$.

Notice that I inline obvious transitions. For example, in step (4), the output state function is On_1, rather than some On_3 that is separately shown to equal On_1.

Figure 5 shows a graphical depiction of the non-interference theorem. Essentially, D and D' are equal under *Save* when the transition from D to D' turns on a layer, makes an arbitrary number of edits on it, and turns it off. This property is crucial for inspiring user confidence when using layers. With non-interference, users can feel confident making arbitrary edits on layers, removing layers, etc., confident that they can always go back to the prior state of the document.

4 Implementing the TCM using regions

The fine, character-level granularity of the TCM makes reasoning clear and simple. A direct implementation of the TCM at the character level may be undesirable, however. First, filtering each character every time the document is rendered may be inefficient: worst-case $O(\#\text{Layers} \times \#\text{Characters})$. Second, implementing some UI features is more natural when manipulating text *regions* rather than individual characters. For example, an interface may place a single marker wherever a sequence of related text is elided. In short, using text regions speeds up rendering and provides a better, higher-level interface for programmatically manipulating layered documents.

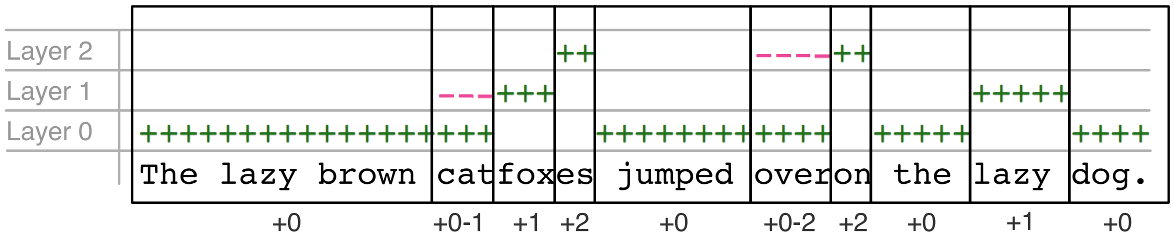


Figure 6: Representing documents using regions. Every box denotes a region. Below each box is the region's tag.

To implement the TCM with regions, we represent the document as a list of tagged regions, rather than a list of tagged characters. Figure 6 shows a pictorial representation of a region-based representation. Instead of tagging individual symbols, adjacent symbols with identical tag are grouped together into a single tagged region. Maintaining the minimal number of regions is preserved throughout editing.

Sketched below are the algorithms for performing text insertions and deletes upon a document representation based on regions. We write $\overrightarrow{c}^{+t-\bar{t}}$ to represent a tagged region, where $\overrightarrow{c}$ is a sequence of characters. We write $\overrightarrow{c}_1 \cdot \overrightarrow{c}_2$ to mean the concatenation of strings $\overrightarrow{c}_1$ and $\overrightarrow{c}_2$, and "c" to mean the singleton string made from character c.

```
insert1($\overrightarrow{c}^{+t'-\bar{t}}$,$t$,$c$,$p$) =
  $\overrightarrow{c}_1$, $\overrightarrow{c}_2$ = findSplit($\overrightarrow{c}$,$p$)
  if $t$ == $t'$ && $\bar{t} = \emptyset$ then
    [$(\overrightarrow{c}_1 \cdot "c" \cdot \overrightarrow{c}_2)^{+t-\bar{t}}$]
  else
    [$\overrightarrow{c}_1^{+t'-\bar{t}}$,$"c"^{+t-\emptyset}$, $\overrightarrow{c}_2^{+t'-\bar{t}}$]

insert($t$,$c$,$p$) =
  $r$,$p'$ = findRegion($p$)
  if $r$ == null then
    addRegion($"c"^{+t-\emptyset}$)
  else
    $rs$ = insert1($r$,$t$,$c$,$p'$)
    replaceAndFuse($r$,$rs$)
```

```
delete1($\overrightarrow{c}^{+t'-\bar{t}}$,$t$,$p$) =
  $\overrightarrow{c}_1$,"$c$",$\overrightarrow{c}_2$ = findChar($\overrightarrow{c}$,$p$)
  if $t$ == $t'$ then
    [$(\overrightarrow{c}_1 \cdot \overrightarrow{c}_2)^{+t'-\bar{t}}$]
  else
    [$\overrightarrow{c}_1^{+t'-\bar{t}}$,$"c"^{+t'-(\bar{t}\cup\{t\})}$, $\overrightarrow{c}_2^{+t'-\bar{t}}$]

delete($t$,$p$) =
  $r$,$p'$ = findRegion($p$)
  if $r$ == null then
    return
  else
    $rs$ = delete1($r$,$t$,$p'$)
    replaceAndFuse($r$,$rs$)
```

The function `insert(`t,c,p`)` inserts character c on layer t at position p. The position p is a position into the user-visible, rendered text. The function `findRegion(`p`)` finds the region at position p. It can be implemented as a linear search over the region list, skipping hidden regions and counting characters in the visible regions. When the desired region is found, it is returned along with p's relative position inside of the region. The function `addRegion` simply inserts the given region to the end of the region list. The r `== null` case only arises when the document is empty. The function `replaceAndFuse(`r`,`rs`)` first inserts the sequence rs in place of the region r in the region list. If rs contains empty regions, they are ignored. It then fuses together adjacent regions with identical tag, both within rs, and at the boundaries between rs and its neighbors in the region list. The function `delete(`t`,`p`)` deletes, on layer t, the character at position p.

The functions `insert1` and `delete1` are the position-aware, region-based analogues of the cases (ins) and (del1), respectively, from Definition 1. The function `findSplit(`$\overrightarrow{c}$`,`p`)` splits the sequence $\overrightarrow{c}$ into two sequences, at position p. The function `findChar(`$\overrightarrow{c}$`,`p`)` is similar, except it returns a triple $\overrightarrow{c}_1$,"c", $\overrightarrow{c}_2$: the text before position p, the character at position p, and the text after p. If the found region belongs to the current layer, the inserts and deletes are performed directly within the region. Otherwise, the region is split in half around position p and a new region is inserted between the two resulting regions. The syntax `[`r_1`,...,`r_n`]` denotes the list containing $r_1, \ldots, r_n$.

Cases (del2) and (del3) from Definition 1 do not have analogues in the region-based implementation because in the implementation there are no regions explicitly marked hidden or dangling. These states are computed directly as functions over regions.

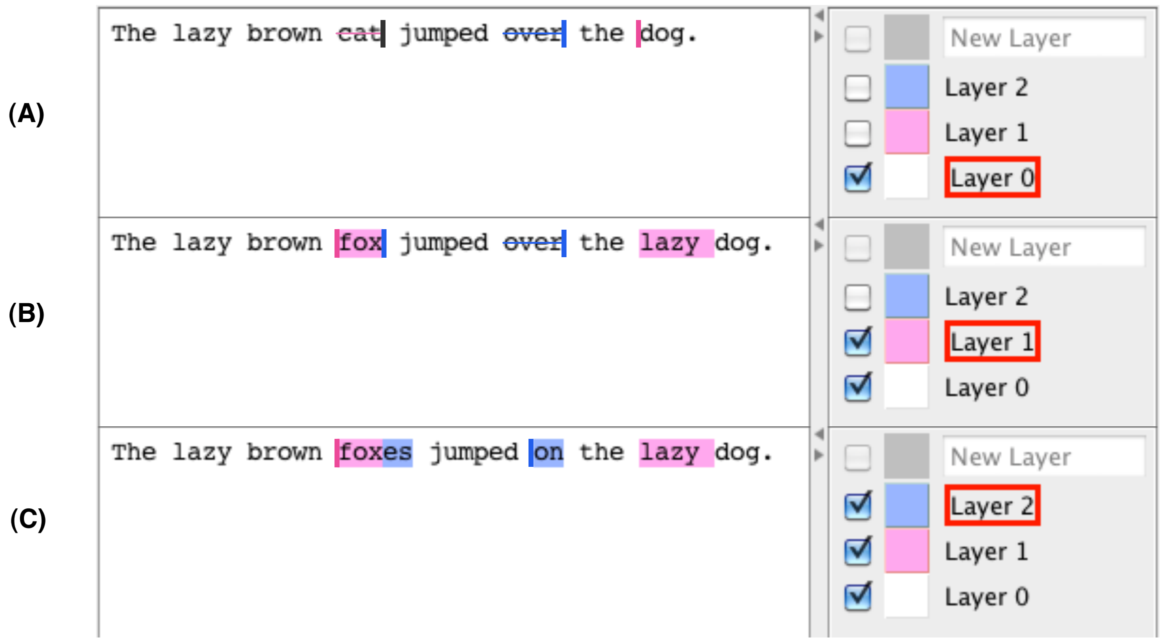
5 User Interface Considerations

So far, we have discussed a basic prototype implementation of layers, its underlying model, and some of its properties. A number of user interface questions have not been considered: What should the editing surface expose to the user? How would the user navigate a layered document? Should some layers be dependent upon one another? The rest of this section gives consideration to each of these questions in turn.

Representing State on the Editing Surface

The underlying representation of a layered document contains all of the inserted and deleted text, properly labeled by the layers taking action upon it. In our examples so far, however, only a subset of the text is shown to the user, depending on the layers on/off state. For example, if a layer is turned on, its inserts are shown and its deletes are hidden. If it is off, its deletes are shown and its inserts are hidden. In the prototype implementation, when a region of text is hidden, there is no indication, from the user's perspective, that text is being elided at that particular location. In some applications, it is desirable to place indicators at the locations where text is hidden. For example, if debugging/tracing code is written on a layer, it is useful to keep that code out of the way when irrelevant (by keeping the layer off), but it is also useful to understand *where* the debugging statements are inserted, to keep an abstract view for how an edit affects the document as a whole.

Figure 7 shows a mockup of Figure 2, annotated with *edit markers*. A *show/hide marker* is either a thin vertical line, the height of one character, that is painted on the background of the editing surface between any two characters. A *delete marker* is a thin horizontal line


(A)

The lazy brown ~~cat~~ jumped ~~over~~ the dog.

(B)

The lazy brown fox jumped ~~over~~ the lazy dog.

(C)

The lazy brown foxes jumped on the lazy dog.

Figure 7: Figure 2, enhanced with edit markers.

that strikes-through a region of text. An *insert marker* is a solid block of color highlighting a region of text. Show/hide markers belonging to a layer are colored in a slightly darkened version of the layer's color, to be visually distinguished from insert markers. So far in our examples, we used only insert markers.

In Figure 7(A), we see delete markers over the words `cat` and `over`, meaning Layer 1 and Layer 2 would delete those two words, respectively, when turned on. We see show/hide markers in three separate locations. A show/hide marker means that either a layer that is *on* deletes text, or a layer that is *off* inserts text, at that location. In Figure 7(B), Layer 1 is turned on. The word `cat` is replaced by a show/hide marker in Layer 1's color. What was previously a show/hide marker is now expanded into the word `fox`. Similarly, the show/hide marker before the word `dog` is now expanded into an insert marker highlighting `'lazy '`. Figure 7(C) proceeds similarly, for Layer 3.

Using markers in this manner presents a fundamental tension, visible in Figure 7(A). Both Layers 1 and 2 insert text after the word `cat`, but only one marker is visible. The tension is this: either we disturb the space-formatting of the text by taking up extra buffer real-estate to display all the markers, or we leave the original formatting alone and limit ourselves to using a fixed amount of space to display a generally unbounded amount of information. In Figure 7, I chose to use black (the color of the text) to denote multiple insertions. A mouse-over action could display all the layers involved, in a pop-up bubble.

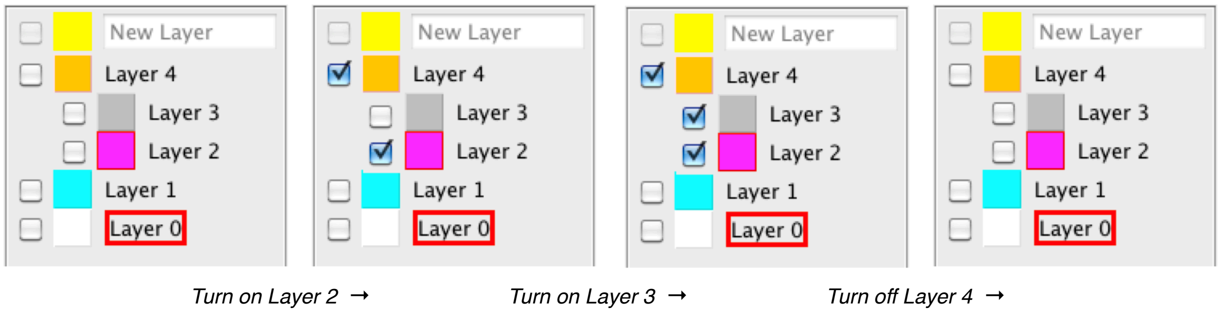


Figure 8: Mockup of a dependency-aware layers control panel. Layers 2 and 3 are shown dependent upon Layer 4 by the level of nesting. When Layer 2 is turned on, its parent Layer 4 is simultaneously turned on. When Layer 4 is turned off, its children, Layers 2 and 3, are simultaneously turned off.

Navigating Layered Document Sets

A layer's edits can be scattered arbitrarily across a document. Depending on the implementation, a layer's edits can span multiple files, scattered arbitrarily across the filesystem. For many tasks, it is useful to quickly find all the edits on a layer. For example, a programmer may want to change all the calls to `foo()` on a layer into calls to `bar()`, due to switching to a new library. Locating the layer's edits is then a crucial prerequisite.

One method for quickly locating a layer's edits within a document is *scrollbar markers*, as implemented in Edit Wear and Read Wear [5] and Rob Miller's LAPIS [9]. The vertical background area of the scrollbar can be annotated with colored markers, each marker denoting an edit on a layer. When the marker is clicked, the scrollbar automatically scrolls to that location, revealing the edit in question.

Another method is to implement a horizontal-split view that shows all the edits on a layer, in sequence and in context. Each layer in the control panel is accompanied by a button that opens that layer in a split view. Each edit in the split view is accompanied by a button that opens the file containing that edit, scrolled to the correct position, for more thorough editing. One version of the layers prototype, implemented by Michael Bayne,[3] features such a split view.

Supporting Dependencies Among Layers

The layers model and prototype implementation do not keep track of dependencies among layers. In some cases, however, it is useful to denote a layer as depending on another layer—meaning that if A depends on B, then A should never be on without B also being on. For example, in Figure 7, it may be intuitive from the user's perspective to declare Layer 2 as dependent upon Layer 1, since the insertion `es` makes sense only in the context of `fox`, which is inserted by Layer 1.

[3]http://cs.washington.edu/homes/mdb

One way to extend layers with user-specified dependencies is to consider layers (previously without structure) to be organized as a set of unordered trees. A layer's children are dependent upon that layer, and a layer depends on all of its ancestors. When a layer is turned off, all of its children are turned off. When a layer is turned on, all of its parents are turned on.

Figure 8 shows how dependencies can be represented in the layers UI. Layers 2 and 3 are shown dependent upon Layer 4 by the nesting level. When either Layer 2 or Layer 3 are turned on, Layer 4, the parent, is automatically turned on. When Layer 4 is turned off, its children are automatically turned off. To add a new dependency, a layer in the control panel can be dragged and dropped onto another layer, thus becoming its child. To remove a dependency, a layer is dragged and dropped *between* two layers at the top level. UI techniques for managing precisely this type of dependency structure are numerous. For example, the bookmark manager in the Safari Web browser.

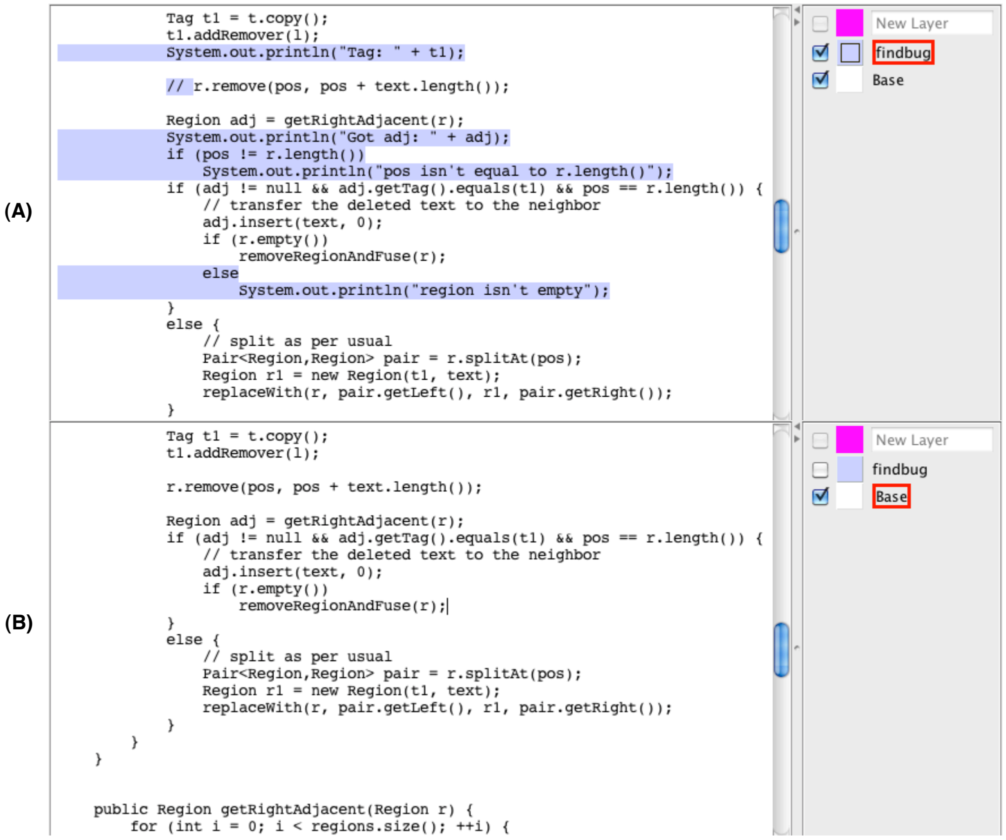
6 Applications to Software Engineering

Graphic designers, using tools like Photoshop and Illustrator, enjoy simple and usable techniques for representing design alternatives, work in progress, and in general working with many versions in varying stages of completion. Software designers, despite taking on the vastly complex task of correct software construction, rely on crude and primitive combinations of tools tools that make basic design tasks difficult, cumbersome, and in many cases impossible. Layers seeks to provide a simple and integrated platform for versioning in the development process. It makes it natural to work with ephemeral, task-only code and to represent development alternatives. In general, it provides a simple way to view and edit versioned text objects.

Delimiting Task-Specific Code
Often, the programmer needs to make a change but does not know what the change should be. A typical example is in the case of a bug. The programmer knows that the bug should be fixed but does not yet know which code structures give rise to the bug. The programmer then engages in program understanding tasks, such as code reading, testing, and running analyses. One easy, natural, and typical program understanding task is to change the program and observe how the changes affect its behavior. The quintessential example is to insert `print` statements at locations of interest.

Although this approach comes naturally to programmers, it suffers from the problem of entangling program code with *task-specific edits*. Task-specific edits are ephemeral and have meaning only with respect to a programming task. When the task is completed, the edits becomes meaningless, and so they should be removed. To find and remove the edits without tool support, programmers resort to leaving behind bread crumbs in the form of comments, working on copies, or searching for and removing the code after the fact.

Figure 9 shows layers being used to delimit task-specific code. In Figure 9(A), task-specific code is written on a layer. When the task has reached its goal, the layer can be



(A)

```
        Tag t1 = t.copy();
        t1.addRemover(1);
        System.out.println("Tag: " + t1);

        // r.remove(pos, pos + text.length());

        Region adj = getRightAdjacent(r);
        System.out.println("Got adj: " + adj);
        if (pos != r.length())
            System.out.println("pos isn't equal to r.length()");
        if (adj != null && adj.getTag().equals(t1) && pos == r.length()) {
            // transfer the deleted text to the neighbor
            adj.insert(text, 0);
            if (r.empty())
                removeRegionAndFuse(r);
            else
                System.out.println("region isn't empty");
        }
        else {
            // split as per usual
            Pair<Region,Region> pair = r.splitAt(pos);
            Region r1 = new Region(t1, text);
            replaceWith(r, pair.getLeft(), r1, pair.getRight());
        }
```

(B)

```
        Tag t1 = t.copy();
        t1.addRemover(1);

        r.remove(pos, pos + text.length());

        Region adj = getRightAdjacent(r);
        if (adj != null && adj.getTag().equals(t1) && pos == r.length()) {
            // transfer the deleted text to the neighbor
            adj.insert(text, 0);
            if (r.empty())
                removeRegionAndFuse(r);
        }
        else {
            // split as per usual
            Pair<Region,Region> pair = r.splitAt(pos);
            Region r1 = new Region(t1, text);
            replaceWith(r, pair.getLeft(), r1, pair.getRight());
        }
    }
}


public Region getRightAdjacent(Region r) {
    for (int i = 0; i < regions.size(); ++i) {
```

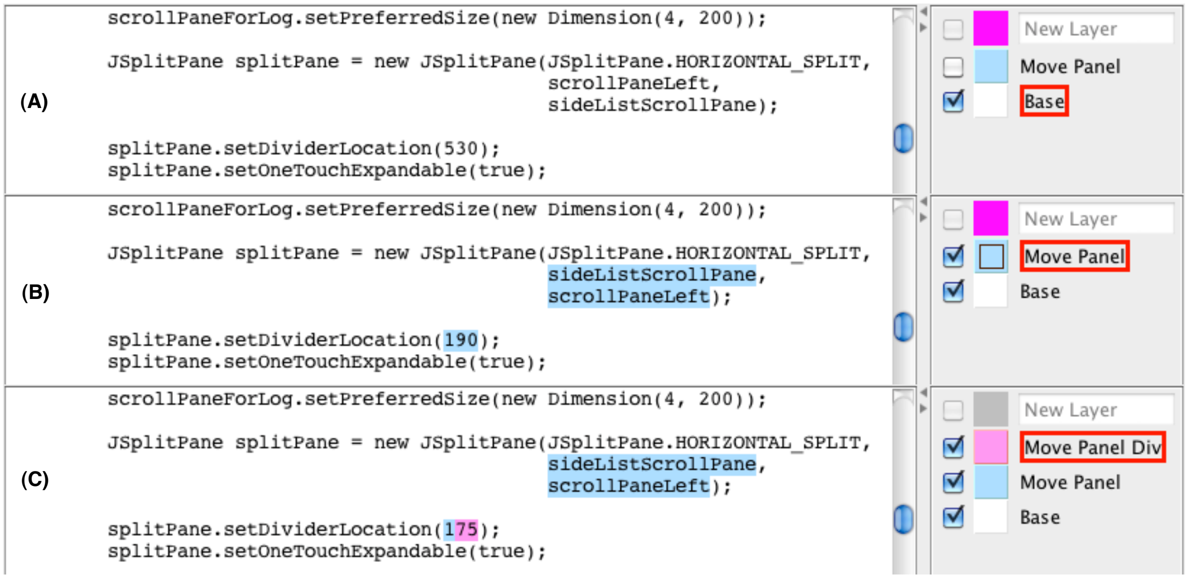
Figure 9: Using layers for ad-hoc debugging. In (A), the programmer writes as-hoc code on the *findbug* layer to corner a behavior. In (B), the *findbug* layer is turned-off or discarded once the behavior has been discovered. Non-interference guarantees that editing actions taken in (A) do not affect (B), the original state of the program.

turned off or discarded, leaving the previous state of the program unchanged (Figure 9(B)). The editing non-interference property of layers guarantees that edits on the debugging layer do not in any way affect the prior state of the program text.

Creating and Maintaining Development Alternatives

When implementing a requirement, programmers may create several implementations before settling on any one in particular. This may be due to not being able to comprehend in advance the tradeoffs of doing things one way or another and wanting to test both to understand the differences. Often, however, programmers must provide several alternatives by client request.

In Figure 10, a hypothetical client requested that several implementations of the layers prototype be provided, including one in which the layers control panel is placed to the right of the editing surface, and one in which the order of the control panel and editing surface



(A)

```
scrollPaneForLog.setPreferredSize(new Dimension(4, 200));

JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                                      scrollPaneLeft,
                                      sideListScrollPane);

splitPane.setDividerLocation(530);
splitPane.setOneTouchExpandable(true);
```

(B)

```
scrollPaneForLog.setPreferredSize(new Dimension(4, 200));

JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                                      sideListScrollPane,
                                      scrollPaneLeft);

splitPane.setDividerLocation(190);
splitPane.setOneTouchExpandable(true);
```

(C)

```
scrollPaneForLog.setPreferredSize(new Dimension(4, 200));

JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                                      sideListScrollPane,
                                      scrollPaneLeft);

splitPane.setDividerLocation(175);
splitPane.setOneTouchExpandable(true);
```

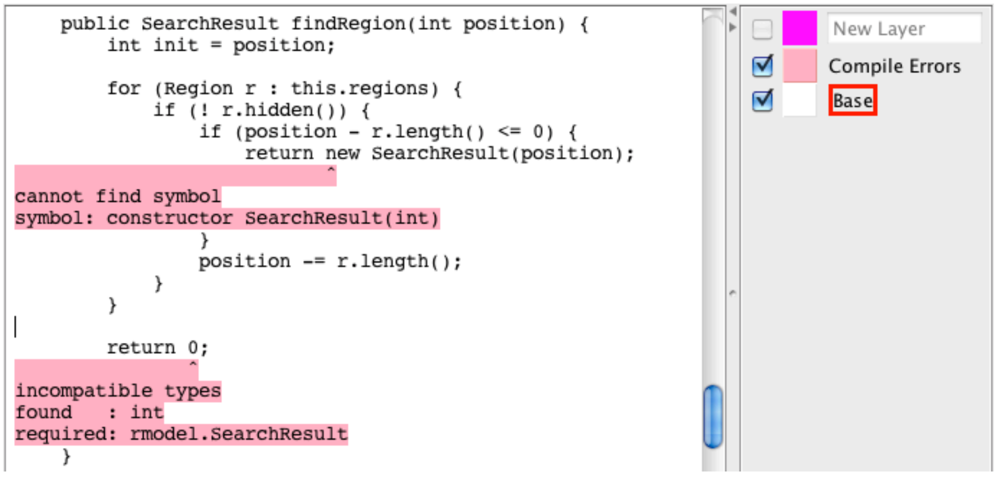
Figure 10: Using layers to express development alternatives. (A) shows code from the layers prototype. (B) shows a version of layers that places the layers control panel to the left of the editing surface. (C) shows a further refinement of (B).

is swapped. When the layer named "Move Panel" is turned on, the resulting code swaps the two panels. Not only does layers enable maintaining two versions of the code, but the relationship between the versions can be examined and understood clearly in the editor. Figure 10(C) shows a further refinement upon "Move Layer" that changes the position of the center divider. The programmer can either compile the three versions separately, or give a live demonstration by flipping between and running the versions directly in the development environment.

This type of code alternative is called a *development alternative* because it is only pertinent in the development process: the choice between the alternatives is not relevant to the user, but only to the development team and client. Other development alternatives may include development-only logging, tracing, and assert statements that are to be excluded from the shipped code, and partially-implemented features that are to be kept out of the way when not being worked on.

Contextual Display of Analysis Results

So far, we have discussed applications of layers in cases that demand maintaining multiple versions of code. Often, programmers deal with tasks that have a naturally versioned structure but this structure isn't expressible in current tools. For example, when running an analysis on a program's source code, the analysis results are positionally linked to the code: the results refer to line numbers and names in the program. Just like a text file along with a positionally-linked `diff` form two versions of the file, so a program and its analysis



```
    public SearchResult findRegion(int position) {
        int init = position;

        for (Region r : this.regions) {
            if (! r.hidden()) {
                if (position - r.length() <= 0) {
                    return new SearchResult(position);
                                           ^
cannot find symbol
symbol: constructor SearchResult(int)
                }
                position -= r.length();
            }
        }

        return 0;
               ^
incompatible types
found    : int
required: rmodel.SearchResult
    }
```

Figure 11: Using code to display compile errors in context. (The technology to automatically generate the error layer is not yet implemented.)

results form two versions: the original program, and the program together with its analysis results. To understand the results, the programmer implicitly considers the program and the analysis results as one object, looking up a program location in order to make sense of a result and vice versa.

Figure 11 shows layers being used to present Java compile errors contextually, placing error messages on a layer, directly at the code position to which it refers. Although the code that automatically generates this layer has yet to be written, all of the required information (line number, character position, and error text) are available in the user-readable error message. Analysis results in general can be presented in this manner, eliminating the awkward manual matching gap between the messages and the program text.

7 Related Work

To our knowledge, layers is the only technology that enables direct manipulation of versions seamlessly into the editing process. Similar in intent to layers is *change tracking*, as implemented in word processors like Microsoft Word, Apple Pages, and OpenOffice. In layers terms, change tracking consists of a single layer whose edits can be individually accepted or rejected. It is best suited for communicating edits to a document among authors and editors; it does not support multiple layers and therefore does not explicitly support editing a document from multiple perspectives and maintaining several interdependent versions.

Juxtapose [4] is an end-to-end program authoring system that supports creating and maintaining program alternatives. Its code editor employs a modified version of the linked editing technique [14] to concurrently edit the parts of the code that are common to all versions. With linked editing turned off, changes remain local to the version being edited.

Juxtapose therefore assumes that the versions being edited are stand-alone copies, and linked editing is used to avoid making the same edit n times to n versions. Layers, on the other hand, maintains a single document in which all versions are combined. Code that is shared among versions exists as a single copy, so there is no need for auxiliary technology to manage this type of duplication. Like the layers implementation, Juxtapose has an analogue to *regions* and uses them to keep linked code blocks aligned for linked editing.

LAPIS [9, 10] is an editing system supporting *lightweight text structure*. LAPIS supports multiple selections of text, simultaneous editing (a precursor to linked editing) of a set of selections, and automated inference of ad-hoc text structure. Its implementation is heavily based on regions and an algebra that supports many operations on regions and region sets. Layers and LAPIS are complementary. LAPIS is concerned with the structure of the underlying text, while layers ignores the structure of the text and manages only the higher-level version structure. LAPIS techniques, in particular selection inference, could help the user identify code that should be on a layer, simultaneously edit regions within a layer, etc.

The C preprocessor (CPP) includes directives (`#if`, `#ifdef`) that can be used for maintaining ad-hoc program versions. The version structure is encoded directly in the program source and version selection is achieved by running the preprocessor over the program text. When using the preprocessor in this manner, there is no notion of editing a version after it has been selected. All versions are edited in the original, combined form, and selection usually only occurs at the time of compilation. Layers is an editing-level technology that does not interfere with the program text and naturally enables focused editing on individual versions. In terms of expressiveness, CPP supports a full language of conditionals (t && t', t || t', etc.), while layers supports only the specialized conditional $+t-\bar{t}$, whose full meaning is evident in Definitions 2 and 6. Layers does not directly support t && t' and t || t', essentially because it does not support text entry on multiple layers (there is only one current layer). Although these conditionals can be simulated in layers by duplicated insertions, it isn't yet clear where they would be useful in the layers paradigm.

Tools like Git, SVN, CVS, Mercurial, etc., enable versioning of text documents. Unlike layers, these tools are disconnected from the editing process. Version commands are issued outside the editor, and individual versions are edited in isolation, unaware of the underlying versioning structure. This process naturally gives rise to conflicts (two versions editing the same line of text), which become evident only at the point of version merging and are resolved manually by the user. Layers has no notion of conflict, as the versions are always merged, always conflict-free. One clear advantage that these tools have over layers is support for multiple developers working in isolation, unaware of each other's activity until the point of the merge. As of yet, there has been no significant consideration for multiple users editing the same layered document concurrently. A baseline possibility is to propagate the edits in real-time, in the manner of Google Wave.[4] Another is to carefully consider the notions of branching and merging in the context of layers.

The Choice Calculus [3] is a recent formal model for software variation. Like the Tagged Character Model, it provides an abstract, mathematical setting for theoretical exploration of

[4] http://www.waveprotocol.org

version management. Its language is powerful, able to succinctly represent complex, multi-dimensional variation. The TCM is much simpler, avoiding to precisely capture notions like *version* and *alternative*, and focusing mainly to ensure that the layered document structure yields a simple editing model. The Choice Calculus seems to focus on providing a structured, sound alternative to CPP, while layers focuses on version integration into the editor.

Aspect-oriented programming (AOP) [7] is a language-based technique for maintaining crosscutting concerns. With AOP, a concern is written separately as an *aspect* and *woven* into the program at points of interest, called *join points*. Join points are described in a regular expression language, so a short description (called a *pointcut*) can capture a large number of join points. Layers can be considered to be an *aspectual editing* technique. The analogue of an aspect is a layer and the analogue of a join point is the location at which an edit is inserted. Layers has no explicit notion of pointcut, so unlike AOP, it cannot describe insertion at multiple positions.

A newer version of AOP that is more closely related to layers is Fluid AOP [6]. Fluid AOP aims to integrate AOP into the development environment, allowing developers to view a program with aspects applied to it. (In non-fluid AOP, aspect weaving happens at compile time or runtime – there is no notion of viewing/editing a woven program.) Unlike Fluid AOP, layers is ignorant to program structure and allows programmers to make arbitrary edits to versions. It is possible that layers is an appropriate editing interface for technologies like Fluid AOP.

Changeboxes [2, 11] is a technique that treats programs as end-to-end versioned, evolving structures. A program is a hierarchy of changeboxes, and a changebox is either empty or a set of edits to an existing changebox. Compiling, running, and interacting with a changebox is treated as independent from the other changeboxes. Changeboxes are similar to layers in that layers are defined in terms of one another, and layers provide a way to gain a multitude of *views* upon a system. Layers is currently ignorant of the underlying system structure, however, and provides no further support beyond multiversion editing. It is unclear whether a changebox-aware text editor exists.

Code bubbles [1] is a recent code editing technique that allows related method-level program fragments to be edited together. In general, the user is responsible for constructing the groups of related methods. Like code bubbles, layers enables grouping of related code (all code on a layer is logically grouped together) and expressing cross-cutting concerns. Unlike code bubbles, layers is edit-based, delimiting not code, but *code transformations* that can be enabled or disabled on demand. It is currently unclear how version management could be combined with code bubbles.

Finally, layers is somewhat related to both programming by example (PBE) and program translation. It is a PBE system in that a layer's editing action is directly specified by the user simply by editing text, without specifying an abstraction. It is a program translation system, similar to AspectJ [8], ARCUM [13], and Twinning [12], in that a layer (a user-specified translation) is applied to a text on demand. Because there is no inference in the editing process, there is no ambiguity in the application of the translation.

8 Conclusions

We have presented the current state of layers, a new text-editing concept that combines a form of lightweight versioning into the editing process. Unlike other versioning techniques, layers exhibits *version coherence*: a layered document is always merged, always conflict-free; therefore, all edits are final and never rejected by a future conflict. Layers seems particularly pertinent to programmers, by providing editing scratch space for exploratory programming, allowing coherent maintenance of multiple development alternatives, and providing a framework for presenting analysis results and other code annotations in context, directly in the source code. The layers editing non-interference property (Theorem 10) guarantees that editing on a layer does not affect the previous state of the document (when the layer is off), and so programmers should feel confident making arbitrarily destructive edits on a layer.

This paper only scratches the surface. We have yet to consider integrating layers into the filesystem, temporal (checkpoint- or commit-based) versioning of layered documents, and the full space of layers *uses*, particularly in the code domain.

References

[1] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. L. Jr. Code bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proc. ICSE*, 2010.

[2] M. Denker, T. Gîrba, A. Lienhard, O. Nierstrasz, L. Renggli, and P. Zumkehr. Encapsulating and exploiting change with changeboxes. In *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, pages 25–49, 2007.

[3] M. Erwig and E. Walkingshaw. Change theory and variation management. http://web.engr.oregonstate.edu/ erwig/ToSC/.

[4] B. Hartmann, L. Yu, A. Allison, Y. Yang, and S. R. Klemmer. Design as exploration: Creating interface alternatives through parallel authoring and runtime tuning. In *Proc. UIST*, 2008.

[5] W. C. Hill, J. D. Hollan, D. Wroblewski, and T. McCandless. Edit wear and read wear. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 3–9, 1992.

[6] T. Hon and G. Kiczales. Fluid aop join point models. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 712–713, 2006.

[7] G. Kiczales and E. Hilsdale. Aspect-oriented programming. In *ESEC/FSE*, 2001.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01: European Conference on Object-Oriented Programming*, 2001.

[9] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *USENIX Annual Technical Conference*, 2002.

[10] R. C. Miller and B. A. Myers. Multiple selections in smart editing. In *Proc. IUI*, 2002.

[11] O. Nierstrasz, M. Denker, T. Gîrba, and A. Lienhard. Analyzing, capturing and taming software change. In *ECOOP Workshop on Revival of Dynamic Languages*, 2006.

[12] M. Nita and D. Notkin. Using twinning to adapt programs to alternative apis. In *Proc. ICSE*, 2010.

[13] M. Shonle, W. G. Griswold, and S. Lerner. Beyond refactoring: a framework for modular maintenance of crosscutting design idioms. In *ESEC/FSE*, 2007.

[14] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *IEEE Symposium on Visual Languages - Human Centric Computing*, 2004.

Appendix: Proof sketches

Theorem 11

1. $\mathit{Valid}(D_1 + D_2; \mathit{On}; t) = \mathit{Valid}(D_1; \mathit{On}; t) \wedge \mathit{Valid}(D_2; \mathit{On}; t)$.
2. $\mathit{Save}(D_1 + D_2) = \mathit{Save}(D_1) \cdot \mathit{Save}(D_2)$.
3. $\mathit{Unfilter}(D_1 + D_2) = \mathit{Unfilter}(D_1) + \mathit{Unfilter}(D_2)$.
4. $\mathit{Filter}(\mathit{On}, D_1 + D_2) = \mathit{Filter}(\mathit{On}, D_1) + \mathit{Filter}(\mathit{On}, D_2)$

Proof Sketch: *Filter*, *Unfilter*, and *Save* are defined inductively, without reasoning about list positions, so if they hold for whole lists, they hold for sublists and vice versa. $\square$

Proof sketch for Thm. 4:
Part (1) goes by induction on D:

- Case $D = \epsilon$. Then $\mathit{Unfilter}(\mathit{Filter}(\mathit{On}, \epsilon)) = \mathit{Unfilter}(\epsilon) = \epsilon$.
- Case $D = c^{+t-\bar{t}} \cdot D'$. $\mathit{Filter}(\mathit{On}, D)$ is either
  - $c^{+t-\bar{t}} \cdot \mathit{Filter}(\mathit{On}, D')$
  - $\mathsf{hidden}(c^{+t-\bar{t}}) \cdot \mathit{Filter}(\mathit{On}, D')$
  - $\mathsf{dangling}(c^{+t-\bar{t}}) \cdot \mathit{Filter}(\mathit{On}, D')$

Either way,

$$\begin{array}{rcll} \mathit{Unfilter}(\mathit{Filter}(\mathit{On}, D)) & = & c^{+t-\bar{t}} \cdot \mathit{Unfilter}(\mathit{Filter}(\mathit{On}, D')) & (\mathit{Unfilter} \text{ defn.}) \\ & = & c^{+t-\bar{t}} \cdot D' & \text{(induction)} \\ & = & D & \end{array}$$

- Other cases not possible, by assumption.

Part (2) is a direct consequence of part (1). □

Proof sketch for Thm. 7:
By induction on D:

- Case ϵ. Follows immediately.
- Case $c^{+t-\bar{t}} \cdot D'$. From the induction hypothesis:

  1. $\mathit{Valid}(D'; \mathit{On}; t) \Rightarrow \mathit{Filter}(\mathit{On}, \mathit{Unfilter}(D')) = D'$.
  2. $\mathit{Filter}(\mathit{On}, \mathit{Unfilter}(D')) = D' \Rightarrow \mathit{Valid}(D'; \mathit{On}; t)$.

  First assuming (1): from the definition of $\mathit{Valid}(D; \mathit{On}; t')$ we get (a) $\mathit{On}(t) \wedge \forall t'' \in \bar{t}. \neg \mathit{On}(t'')$ and (b) $\mathit{Valid}(D'; \mathit{On}; t')$. Then,

$$\begin{array}{rcll} \mathit{Filter}(\mathit{On}, \mathit{Unfilter}(D)) & = & \mathit{Filter}(\mathit{On}, c^{+t-\bar{t}} \cdot \mathit{Unfilter}(D')) & \text{by } \mathit{Unfilter}(-) \text{ defn.} \\ & = & c^{+t-\bar{t}} \cdot \mathit{Filter}(\mathit{On}, \mathit{Unfilter}(D')) & \text{by (a) and } \mathit{Filter}(-,-). \\ & = & c^{+t-\bar{t}} \cdot D' & \text{by (1) and (b)} \\ & = & D & \end{array}$$

  Going the other direction, we have:

$$\begin{array}{rcll} \mathit{Filter}(\mathit{On}, \mathit{Unfilter}(D)) & = & D & \text{top-level assumption} \\ \mathit{Filter}(\mathit{On}, \mathit{Unfilter}(c^{+t-\bar{t}} \cdot D')) & = & c^{+t-\bar{t}} \cdot D' & \text{ditto} \\ \mathit{Filter}(\mathit{On}, c^{+t-\bar{t}} \cdot \mathit{Unfilter}(D')) & = & c^{+t-\bar{t}} \cdot D' & \mathit{Unfilter}(-) \text{ defn.} \\ k \cdot \mathit{Filter}(\mathit{On}, \mathit{Unfilter}(D')) & = & c^{+t-\bar{t}} \cdot D' & \mathit{Filter}(-,-) \text{ defn.} \end{array}$$

  Expanding the standard definition for equality over lists, we have (a) $k = c^{+t-\bar{t}}$ and (b) $\mathit{Filter}(\mathit{On}, \mathit{Unfilter}(D')) = D'$. Inspecting the definition for $\mathit{Filter}(-,-)$, (a) can be true only if (c) $\mathit{On}(t) \wedge \forall t'' \in \bar{t}. \neg \mathit{On}(t'')$. Applying (b) to (2), we get $\mathit{Valid}(D'; \mathit{On}; t')$. This, together with (c), proves $\mathit{Valid}(D; \mathit{On}; t')$.

- Case $\mathsf{hidden}(c^{+t-\bar{t}}) \cdot D'$. Similar to the above.
- Case $\mathsf{dangling}(c^{+t-\bar{t}}) \cdot D'$. Similar to the above. □

Proof sketch for Thm. 8:
By induction on n.

- Case 0. Vacuous.

- Case n. By induction, $n-1$ actions have been performed and $\mathit{Valid}(D_2; \mathit{On}_2; t_2)$. We need to show that if $D_2; \mathit{On}_2; t_2 \vdash A \rightarrow D_3; \mathit{On}_3; t_3$, then $\mathit{Valid}(D_3; \mathit{On}_3; t_3)$. With the aid of Theorem 7, it suffices to show $\mathit{Filter}(\mathit{On}_3, \mathit{Unfilter}(D_3)) = D_3$. We exhaust A:

  - Case (on). $D_3 = \mathit{Filter}(\mathit{On}_3, \mathit{Unfilter}(D_2))$. By Theorem 4 part (1), $\mathit{Unfilter}(\mathit{Filter}(\mathit{On}_3, \mathit{Unfilter}(D_2))) = \mathit{Unfilter}(D_2)$, so $\mathit{Unfilter}(D_3) = \mathit{Unfilter}(D_2)$.
  - Case (off). Same as (on).
  - Case (switch). The valid input state remains unchanged.
  - Case (edit). $D_3 = \mathit{Edit}(e_{t_2}, D_2)$ and $\mathit{On}_3 = \mathit{On}_2$. We therefore need to show $\mathit{Filter}(\mathit{On}_2, \mathit{Unfilter}(\mathit{Edit}(e_{t_2}, D_2))) = \mathit{Edit}(e_{t_2}, D_2)$. Noticing that $\mathit{On}_2(t_2) =$ true, per our assumption that $S_2 \neq \mathsf{ERROR}$, we examine the cases for $\mathit{Edit}(e_{t_2}, D_2)$:
    - Case (ins). $\mathit{Edit}(\text{insert}_{t_2}(c), D_2) = D + [c^{+t_2-\emptyset}] + [k] + D'$. By induction, we know $\mathit{Valid}(D + [k] + D'; \mathit{On}_2; t_2)$, and from Theorem 11, we have $\mathit{Valid}(D; \mathit{On}_2; t_2)$ and $\mathit{Valid}([k] + D'; \mathit{On}_2; t_2)$. It remains to be shown that $\mathit{Valid}([c^{+t_2-\emptyset}]; \mathit{On}_2; t_2)$.

$$\begin{array}{rcll} \mathit{Filter}(\mathit{On}_2, \mathit{Unfilter}([c^{+t_2-\emptyset}])) & = & \mathit{Filter}(\mathit{On}_2, [c^{+t_2-\emptyset}]) & (\mathit{Unfilter} \text{ defn.}) \\ & = & [c^{+t_2-\emptyset}] & (\mathit{On}(t_2) = \text{true}) \end{array}$$

      By Theorem 7, $\mathit{Valid}([c^{+t_2-\emptyset}]; \mathit{On}_2; t_2)$.
    - Case (del1). $\mathit{Edit}(e_{t_2}, D_2)$ is either $D + D'$ or $D + [\mathsf{hidden}(c^{+t-\{t_2\}})] + D'$. The first case follows from induction and Theorem 11. In the latter case, we have

$$\begin{array}{rcl} \mathit{Filter}(\mathit{On}_2, \mathit{Unfilter}([\mathsf{hidden}(c^{+t-\{t_2\}})])) & = & \mathit{Filter}(\mathit{On}_2, [c^{+t-\{t_2\}}]) \\ & = & [\mathsf{hidden}(c^{+t-\{t_2\}})] \end{array}$$

      the latter due to $\mathit{On}_2(t_2)$ being true. So $\mathit{Valid}([\mathsf{hidden}(c^{+t-\{t_2\}})]; \mathit{On}_2; t_2)$. The rest follows from induction and Theorem 11.
    - Cases (del2) and (del3) follow directly from the induction hypothesis, since the document remains unchanged. □

Proof sketch for Thm. 10:
By induction on n.

- Case 0. If zero edits are performed, then $D_1 = D_2$, since nothing happens at step (3). By inspecting step (4), $D_3 = \mathit{Filter}(\mathit{On}_1, \mathit{Unfilter}(D_2)) = \mathit{Filter}(\mathit{On}_1, \mathit{Unfilter}(D_1))$. We need to show that $\mathit{Save}(D_1) = \mathit{Save}(\mathit{Filter}(\mathit{On}_1, \mathit{Unfilter}(D_1)))$. But because we assume $\mathit{Valid}(D_1; \mathit{On}_1; t_1)$, we have $D_1 = \mathit{Filter}(\mathit{On}_1, \mathit{Unfilter}(D_1))$.

- Case n. By induction, $n-1$ edits have been performed and upon performing steps (4) and (5), $\mathit{Save}(D_1) = \mathit{Save}(D_3)$, or, by inspecting step (4),

$$\mathit{Save}(D_1) = \mathit{Save}(\mathit{Filter}(\mathit{On}_1, \mathit{Unfilter}(D_2)))$$

Between steps (3) and (4), we perform the nth edit,

$$D_2; On_2; t_2 \ \vdash \ \text{Edit } e \rightarrow \mathit{Edit}(e_{t_2}, D_2); On_2; t_2$$

Then, step (4) gives

$$\mathit{Edit}(e_{t_2}, D_2); On_2; t_2 \ \vdash \ \text{Turn off } t_2 \rightarrow \mathit{Filter}(On_1, \mathit{Unfilter}(\mathit{Edit}(e_{t_2}, D_2))); On_1; t_2$$

We need to show:

$$\mathit{Save}(D_1) = \mathit{Save}(\mathit{Filter}(On_1, \mathit{Unfilter}(\mathit{Edit}(e_{t_2}, D_2))))$$

We case on the edit:

- Case (ins). $\mathit{Edit}(e_{t_2}, D_2) = D + [c^{+t_2-\emptyset}] + [k] + D'$. By induction, we have $\mathit{Save}(D_1) = \mathit{Save}(\mathit{Filter}(On_1, \mathit{Unfilter}(D + [k] + D')))$. Therefore, we must show $\mathit{Save}(\mathit{Filter}(On_1, \mathit{Unfilter}([c^{+t_2-\emptyset}]))) = \epsilon$. Because $On_1(t_2) = \text{false}$, *Filter* will yield $[\mathsf{hidden}(c^{+t_2-\emptyset})]$, which *Save* takes to ϵ.
- Case (del1). $\mathit{Edit}(e_{t_2}, D_2)$ is either (a) $D + D'$ or (b) $D + [\mathsf{hidden}(c^{+t'-\bar{t}\cup\{t_2\}})] + D'$. By induction, $\mathit{Save}(D_1) = \mathit{Save}(\mathit{Filter}(On_1, \mathit{Unfilter}(D + [c^{+t'-\bar{t}}] + D')))$.
  When (a) is the case, we essentially need to show

  $$\mathit{Save}(\mathit{Filter}(On_1, \mathit{Unfilter}([c^{+t'-\bar{t}}]))) = \epsilon$$

  We know $t' = t_2$ and so $On_1(t') = \text{false}$. Therefore, $\mathit{Filter}(On_1, [c^{+t'-\bar{t}}])$ yields either a hidden or dangling symbol which is taken to ϵ by *Save*.
  When (b) is the case, we need to show

  $$\mathit{Save}(\mathit{Filter}(On_1, \mathit{Unfilter}([c^{+t'-\bar{t}}]))) = \mathit{Save}(\mathit{Filter}(On_1, \mathit{Unfilter}([\mathsf{hidden}(c^{+t'-\bar{t}\cup\{t_2\}})])))$$

  or

  $$\mathit{Save}(\mathit{Filter}(On_1, [c^{+t'-\bar{t}}])) = \mathit{Save}(\mathit{Filter}(On_1, [c^{+t'-\bar{t}\cup\{t_2\}}]))$$

  We know $t' \neq t_2$ and $t_2 \notin \bar{t}$. Theorem 8 and Definition 6 give $On_2(t') = \text{true}$ and $\forall t'' \in \bar{t}.\neg On_2(t'')$. Because On_1 and On_2 coincide except at argument t_2, $On_1(t') = \text{true}$ and $\forall t'' \in \bar{t}.\neg On_1(t'')$. Also, we know $On_1(t_2) = \text{true}$. Therefore,

  $$\mathit{Save}(\mathit{Filter}(On_1, [c^{+t'-\bar{t}}])) = \mathit{Save}(\mathit{Filter}(On_1, [c^{+t'-\bar{t}\cup\{t_2\}}])) = \text{“}c\text{''}$$

- Cases (del2) and (del3) follow from induction, since the document remains unchanged. □

Omitted details involve invoking Theorems 7, 8, and 11 as needed.