Generalizing Operational Transformation to the Standard General Markup Language

Author,
Faculty,
University, State, Country
address@domain

ABSTRACT

In this paper we extend operational transformation to support synchronous collaborative editing of documents written in dialects of SGML (Standard General Markup Language) such as XML and HTML, based on SGML's abstract data model, the grove. We argue that concurrent updates to a shared grove must be transformed before being applied to each replica to ensure consistency. We express grove operations as property changes on positionally-addressed nodes, define a set of transformation functions, and show how to apply an existing generic operational transformation algorithm to achieve this. This result makes synchronous group editing applicable to the modern Web.

KEYWORDS: synchronous collaborative editing, operational transformation, SGML, XML, groves, computer supported cooperative work

Introduction

In the web, there is a trend towards multiple authors interacting, in contrast to the traditional model of one author publishing to many readers. In group editing, there is a trend away from documents of limited general utility (idiosyncratic formats or limited expressiveness) towards mainstream rich document formats. The next two subsections explore these two arguments.

In this paper, we argue that the intersection of these two trends is synchronous collaborative editing of instances of a structured, metadata-rich data model: trees. The expansion of operational transformation into this new problem space yields more useful forms of collaboration across the Web.

Tree data models need to be multiple-user

The World Wide Web is a fundamentally distributed system. Current abstract models for XML and HTML [20] and their parent language SGML [7] are composed of a single stream of events applied to a mutable tree of objects. These are suitable for the common case of one author generating operations and many readers passively viewing the published document.

Increasingly this polarization of roles is fading. The vision of the Semantic Web [1] requires that an agent is able to create, update, and peruse information freely to achieve its ends. The more concrete Web Services architecture [3] from IBM has no entrenched notion of "author" and "reader", describing interactions across the Web in terms of endpoints residing on peer sites. Even in traditional Web pages, the distinction between author and reader is blurring. The Web Distributed Authoring and Versioning [6] extensions to HTTP are designed to support several authors in collaboration over a Web resource. Wikis [4] are normal HTML pages that have hyperlinks enabling any reader to edit them at any time. Wikis such as the Portland Pattern Repository serve as places where a community stores and collaborative generates knowledge. The Python, Perl, Tcl, Ruby, and Haskell communities are also prominent among the several hundred wikis now in existence according to Google.

By discarding the assumption of a single source of operations to be applied to a single instance of the document, useful collaborative tools can be constructed that leverage the Web. These tools support coarse-grained collaboration, in that different authors can edit different HTML files concurrently. They do not support fine-grained collaboration, where two authors can work on different parts of the same HTML document seamlessly, because they have no way to maintain consistency within a shared instance of the data model. They do not support synchronous collaboration, where each author is aware of the rest of the group's activity as it happens, because they have no notion of other instances of the document which should reflect remote edits.

The coarse-grained and asynchronous collaboration available on the Web is valuable. Upgrading it to fine-grained, synchronous collaboration would only enhance the flexibility that makes it valuable. To achieve that, we must be able to collaborate *within* a document, and therefore need a data model that can maintain consistency in the face of multiple authors. In this paper we propose such a model, by adding concrete addressing and operations to the SGML grove, and defining how to apply multiple streams of these operations to a replicated instance of that grove.

Multi-user linear data models need to be trees

Synchronous collaborative editing is useful because it allows multiple authors to interact through a shared document. The state of the art in synchronous collaborative text editing is operational transformation. The major existing operational transformation editors [19] [14] [12] use an abstract data model of a single linear sequence of content data. Such a flat sequence can model flat text. It cannot model formatted text, because that must contain metadata about font, weight, colour for various nesting subsections of the document. It cannot model any non-trivial vector graphic, as graphical objects must be composable out of other shapes, and various levels of nesting must have content data about shape, size, color, position associated with them. It cannot model SGML, because the elements of the document defined by tag pairs must be able to nest, and the elements must have data from the tag properties associated with them. Therefore it cannot model HTML nor XML, which are dialects of SGML. Therefore, the techniques defined on this data model cannot be applied to the bulk of the content on the Web. This constraint needs to be removed to allow synchronous collaborative editing to be relevant to the Web. The way to remove it is to adopt a more general abstract data model and define techniques for operational transformation based on that model. Above we have argued that the Web represents a large collection of shared documents through which people interact, so the tree common to the Web is a very good abstract data model to adopt.

A Tree-based Data Model

Consider a document consisting of the plain text sentence:

> The quick brown fox jumped over the lazy dog lying next to the top of the cliff, and regretted it briefly.

which is a single object containing a logical sequence of characters (Figure 1) in a normal text editor.

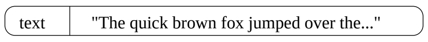

Figure 1: A paragraph as a single sequence of characters

Compare this with the information expressed by

> The quick brown fox jumped over the lazy dog *lying next to the top of the cliff,* and regretted it briefly.

There is still the content data (the text) but there is also data about the content data (metadata) - some parts of the sentence are emphasized, and others are not, conveying meaning to the other authors. Instead of one element of data, there are three: “The quick brown fox jumped over the” with no emphasis, “lying next to the top of the cliff” with emphasis, and “and regretted it briefly” with no emphasis. This is shown in Figure 2.

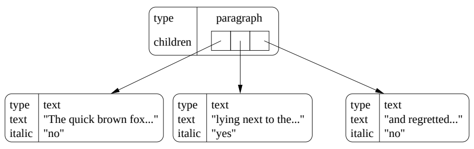

Figure 2: A paragraph as several sequences of characters

After some more editing by the various authors, Figure 3 shows

> The quick brown fox jumped over the lazy dog *lying next to the* ***top*** *of the cliff,* and regretted it briefly.

The second element of the document is itself containing three elements as its content data (“lying next to the”, “top”, “of the cliff”), each with their own metadata

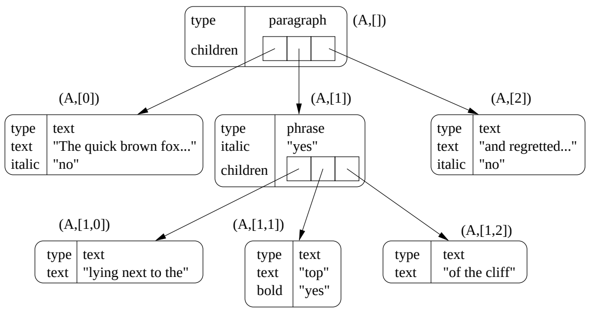

Figure 3: A paragraph as several sequences of characters

The data model this exemplifies is a tree of nodes. A node is a collection of properties, and has (according to its type):

- which property keys (strings) it contains, and what values each property may have,
- at most one *subnode property* which holds a sequence of subordinate nodes, and
- exactly one *content property* which holds the content data of the node (the text, in the example).

Our proposed set of property types are scalars (integers, real numbers, characters), sequences (strings, vectors of values indexed by number), and nodes. An edge exists in the tree from each node containing such a *nodal* property to the node in the property value. *Subnode edges* are edges due to the subnode property. The subnode edges link the nodes into a *subnode tree*. A *child property* is a subnode property that is also the content property for its node. *Child edges* arising from child properties represent parent-child relations between nodes. The child edges form at least one *content tree* out of the grove nodes Where there is only one content tree, the content tree root is also the *grove root*, or root of the subnode tree. Nothing rules out having several content trees, hence the name *grove*.

The tiny grove in Figure 3 has a single content tree. The root is an element of type “paragraph”. Its subnode property is named “children”. The elements held in this property’s value (some text, a phrase, and some more text) are the content data of the paragraph, so this is also a children property and they are child elements. The first and last children are leaf elements which hold plain text in their “text” property and metadata describing presentation in their “italic” property. The “text” property is the content property for a text node, but it is not a children property because it does not contain other nodes. The second node of the root is of type “phrase”. It has three children of its own, all “text” elements, each holding some content data. The “italic” property in their parent affects them, but each adds its own metadata - the middle one has a value of “yes” for its “bold” property.

The data structure described above is essentially[1] the grove defined in the Standard General Markup Language [7] and HyTime [9] standards as the abstract data model for SGML. Where the distinction matters, we will call it a *generalized grove*. The same concepts are used in the Document Object Model [20] for XML, the RELAX NG [13] schema language for XML, and the DSSSL [8] transformation language for SGML.

In our architecture, each author’s editor holds a private replica of the common grove at their own site. Each operation is applied by the site whose author generated it as a *local operation*, and then broadcast to the other sites, who transform it and apply it as a *remote operation* on their groves to maintain consistency.

Addressing a Grove

The messages between sites need to specify which nodes are to be affected. For this purpose a consistent addressing scheme is needed. Such an addressing scheme needs the following characteristics:

1. Addressability of locations as well as nodes. It is essential to be able to discuss locations that a node does not yet occupy, such as when placing a node.
2. Light weight. Addresses which occupy less space and algorithms which are less complex and costly are preferred.
3. Site independence. The same address should refer to the same node on all sites, and dereferencing or generating an address should not require the exchange of messages, which costs time, bandwidth and complexity.
4. Domain independence. Schemes that do not rely on assumptions peculiar to one notation or one kind of grove are preferred.

[1] SGML groves do not support real-valued numbers. More importantly, single-author groves have exactly one subnode tree. Holding parts of the grove being worked on concurrently can require more than one. SGML groves are also not replicated.

There are two basic approaches: name each node and address them by name, or choose certain nodes as landmarks and address nodes by their position with respect to these landmarks. For small groves, naming all nodes is workable. Lightweight names can be chosen independent of the contents of the node or the local structure of the grove and therefore of the site or domain[2]. We observe two problems with this approach.

The minor problem is that a location must be referred to by the label of the node that occupies it. If a node has no children, but we wish to refer to the location where its first child would be, how can this be accomplished? For certain cases, such as Figure 4, an operation can refer to landmarks. Here

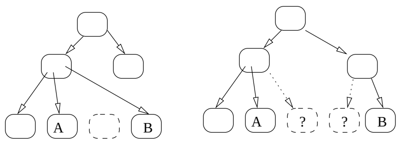

Figure 4: Why pure naming is ambiguous

an author is attempting to create a node between the existing nodes A and B, and specifies them to address that location, illustrated on the left. This only works if no *other* author moves one of the landmarks in the interval between the first author issuing the operation and the other sites receiving it. The right-hand tree of Figure 4 shows such a case. Is the new node next to A or B? It can be shown that a single landmark is insufficient - there is no unique location “to the left of B” or “to the right of A”. So long as order is important[3] this *landmark problem* will occur.

The major problem is that non-trivial hypertexts such as wikis [4] and online books have many nodes. To honour a remote operation, a site must translate the addresses it gives into useable references to actual nodes. When this information is not in the names, it must keep a table mapping names to addresses. This table costs one record of space for each node named, which scales poorly to large projects with multiple authors.

[2] Let the node N be the n^{th} node in the grove created by the operation O, which was the m^{th} operation generated by site S. Then the tuple (S, m, n) is a suitable name for N. Such a name is lightweight, as S, m, and n are cardinal numbers. For clarity, we will abbreviate each unique node name as a letter of the alphabet; instead of $(4, 2, 3)$ we will write D.

[3] To see that order is important in a document, consider a book chapter whose paragraphs are in an arbitrary order that varies between sites.

Therefore, we adopt a positional addressing scheme where the minimum of nodes have names. All others are anonymous, described by the path from their nearest named ancestor. A grove address N for a node is the name of this ancestor, plus a vector of indices describing the path between the ancestor and it. We will write $\langle N \rangle$ to mean the name of the ancestor which is at the root of the path N and $N[i]$ for the i^{th} element of the path vector. $N[0]$ is the position of the first node along the path in the children property of the path root. $N[1]$ is the position of the second node along the path in the children property of the first node in the path. $N[i]$ is the position of the $(i+1)^{th}$ node of the path in the children property of the i^{th} node of the path. Alternately, a complete grove address is written $(R, [i_1, i_2, \ldots])$. The mapping between an anonymous node and its address is stored in the grove itself, instead of bulking up the lookup table. We choose to give names only to the root of each subnode tree, which cannot be described as another node's child. For a concrete example, Figure 3 shows the grove with the addresses next to each node.

Operating on a Grove

The fundamental operations we support on groves need to meet the criteria:

1. *Completeness* - any valid grove can be built from any other valid grove by applying a sequence of fundamental operations.

2. *Disambiguity* - each fundamental operation (and its operands) contains enough information to execute correctly by itself.

3. *Parsimony* - useful tasks are expressible efficiently in terms of these fundamental operations.

The set of operations `insert`, `delete`, and `change` fits these criteria. The first two are *structural* - they change the structure of the grove. The latter is a *mutation* - it changes the data without changing the grove structure.

Insertion The operation $\mathrm{insert}(\mathrm{N}, \mathrm{n}, \mathrm{M}, \mathrm{T}, \ldots)$ adds a node to a subnode tree. If a node with name M exists in the grove, it is inserted in the n^{th} position of the children property of the node whose grove address is N - it becomes the n^{th} child of N. If no such node exists, a node of type T is created and inserted as the n^{th} child of N, with initial values supplied by any other operands.

Deletion The operation $\mathrm{delete}(\mathrm{N}, \mathrm{n}, \mathrm{M})$ excises the n^{th} node from the children property of the node whose grove address is N when executed.

When the base of a branch is removed from a tree, the rest of the branch follows. To see why, consider the grove shown on the left of figure 5, where nodes of type D may only have children of type A or C, and are forbidden children of type B.

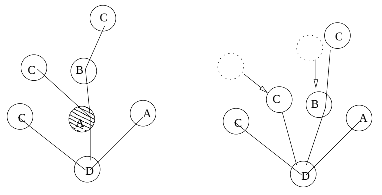

Figure 5: Why deleting single nodes is undesirable

When the shaded node is deleted, its descendants must either be removed with it or replaced in the tree. If we put the deleted node's children in its stead, the resulting grove is no longer valid. Rather than devise ad-hoc, application-specific rules forbidding certain deletions, we consistently remove the entire branch.

Given this, we must consider the possibility that another author was working on the branch, and so there might be remote operations in transit that refer to nodes in the branch. We may not destroy the nodes concerned until we can prove this is not the case, so we will save the excised as another subnode tree. The subnode tree root will need a label, supplied as the M operand in an delete. As a bonus, saving the pruned branch as a separately labelled subnode tree makes implementing cut-and-paste or other movement of existing parts of the grove trivial, but that is beyond the scope of this paper.

Mutation The operation $\mathrm{change}(\mathrm{N}, \mathrm{k}, \mathrm{f}, \ldots)$ changes the content data or metadata in a grove node. Before it is executed, the property whose key is k on the node of the grove whose address is N has the value v. After the change is executed, it has the value $f(v)$. The function f, which takes the old value and returns the new value, is a *transition function*.

In principle a transition function can be arbitrary. For SGML, HTML, or XML, property values are strings and string editing is of most immediate value. An appropriate set of transitions can be found in [19], which defines insert and delete on strings and gives a set of inclusion and exclusion transformations. The fourth and succeeding operands supply any other parameters the transition function needs - insertion requires a string to insert and an offset within the string to insert it at, for example.

Strictly speaking, the minimal complete set of operations is node creation and deletion. Mutating a node *could* be expressed as deleting that node and inserting a new node in the same place with the changed property value. This is not parsimonious because a node with n descendants would require $(n + 2)$ operations to mutate it. All $(n + 1)$ nodes in the branch would have to be reconstructed, which takes $(n + 1)$ insertions. A concurrent operation on a node after its old incarnation was deleted would be lost, as there is no way to know the old and new nodes are the same, and the old address would either be invalid or valid but point to the wrong node during the reconstruction.

Vector Operational Transformation

Background

A consistency model We adopt a consistency model due to Sun et al [19]. A set of groves is *consistent* iff it exhibits the three properties:

1. *Convergence* When all sites have executed the same set of operations, the structures of corresponding groves and the value of properties on corresponding nodes of each grove are identical.
2. *Causality preservation* For each pair of operations O_a, O_b, if $O_a \rightarrow O_b$, then O_a is seen before O_b by all authors - effects follow causes. The standard definition of $\rightarrow$ is due to Lamport [11]. Given two operations O_a and O_b originating from sites S_a and S_b, we write $O_a \rightarrow O_b$ (read "*O_a happens before O_b*") iff

   (a) $S_a \neq S_b$ and O_a was executed at S_b before O_b was created, *or*

   (b) $S_a = S_b$ and O_a was generated before O_b was generated, *or*

   (c) $\exists O_c$ such that $O_a \rightarrow O_c$ and $O_c \rightarrow O_b$.

   We say $O_a \| O_b$ (read "*O_a concurrent with O_b*") iff neither $O_a \rightarrow O_b$ nor $O_b \rightarrow O_a$.
3. *Intention preservation* At each site where a given operation O is executed, the execution effect is same as the original intention of O at the site it was generated. Following Sun's approach, we define the intention of O as the syntactic effect of O at the site of its origin.

We enforce causality by maintaining a vector clock at each site. The vector clock at site i, T_i, is a vector of integers such that $T_i[j]$ is the number of operations from site j that have been executed so far at site i. We speak of an operation O as *causally ready* at a site i if that site's vector clock indicates

1. It is the next operation to be executed from its site of origin.
2. Each operation $X \rightarrow O$, has already been executed here.

If an operation arrives from another site and is not causally ready, it is delayed until the missing operations have arrived and been executed. Local operations are always causally ready and thus never delayed.

Path comparison In the following sections, we will need to compare two paths and determine any areas of congruence between them. The following defines this path comparison:

```
$\mathrm{compare}(\mathrm{N_a}, \mathrm{N_b})$
    if ($\langle N_a \rangle = \langle N_b \rangle$)
        for $i \leftarrow 0 \ldots (|N_a| - 1)$
            if ($i = |N_b|$)
                return PREFIX($i$)
            elif ($N_a[i] \neq N_b[i]$)
                return DIFFERENT
        if ($|N_a| = |N_b|$)
            return SAME
        else
            return SUFFIX
    else
        return DIFFERENT
```

It takes two grove addresses, compares them, and returns one of four verdicts:

1. The two paths are DIFFERENT branches from the same path root, or run from DIFFERENT roots. No operation at N_b could affect the address of the node at N_a, or vice-versa. No operation at N_b could interact with any operation at N_a, or vice-versa.
2. The first and second paths point to the SAME node. An operation at N_b cannot affect the address of the node at N_a, only the addresses of N_a's children. An operation at N_b might still interact with an operation at N_a on *the same property*, but that will only require adjusting the parameters of the operation at N_a, not the addresses.
3. The second path (N_b) is a PREFIX of the first path (N_a), meaning the N_b's vector forms the first i elements of N_a's path vector and the node at N_b forms part of the path of N_a. A structural operation at N_b may therefore affect the path vector of the node at N_a. Which component of that vector is affected depends on i, so it is passed as part of the result, for example PREFIX(42).
4. The second path (N_b) is a SUFFIX of the first path (N_a). This is the dual of the previous case. An operation at N_b will not affect the address of the node at N_a.

Consider the two-tree grove of Figure 6. The paths to the nodes at addresses $(A, [0, 1, 0])$ and $(A, [0, 2])$ branch off from each other at node $(A, [0])$, so compare as DIFFERENT. The paths to the nodes at addresses $(A, [0, 1, 1])$ and $(B, [0, 0])$ start at different roots, so they also compare as DIFFERENT.

The address $(A, [0, 1, 1])$ compares out as a PREFIX(3) of $(A, [0, 1, 1, 0, 1])$, as the paths share the same root and the first 3 nodes in the path. The address $(B, [0, 0, 0])$ compares out as a SUFFIX of the node address $(B, [])$, as does $(B, [0, 0, 1, 0])$, but $\mathrm{compare}((\mathrm{B}, [0, 0, 0]), (\mathrm{B}, [0, 0, 1, 0]))$ yields DIFFERENT.

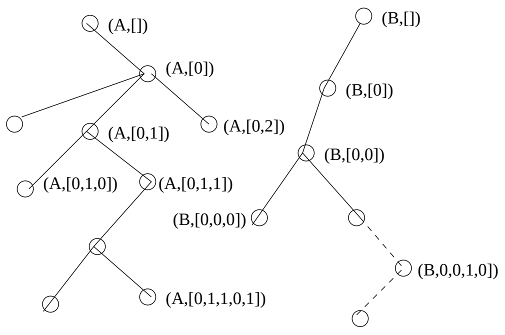

Figure 6: Comparing paths in an example grove

Context To discuss preserving operation intentions, we must introduce Sun and Ellis' notion of *operation context* [17]. The context applying to a document state is the sequence of operations that were applied to bring it from the initial state to the current state. In particular, we distinguish two contexts that are important with respect to an operation O. The *original definition context* of O is the one that describes the document the original user saw when they issued the operation. We write $O_a \sqcup O_b$ if the current definition context of O_a is equivalent to the current definition context of O_b. We write $O_a \mapsto O_b$ if O_b's current definition context is (O_a's current definition context) + O_a ; O_b has O_a immediately preceding it. O_a is said to be *context preceding* O_b. The *execution context* of O is the one that describes the document when O is executed on it. In single-user editors, O is executed at the time is it generated, so the two contexts always match and the effect of O is always as intended. In editors with multiple replicas of the grove, as here, this is only true at the site which generated O. To achieve the intended effect of O at other sites, we must check whether O's current definition context D matches the current execution context H[4]. If we only try to execute O when it is causally ready, then all operations in H will either have happened before O and are already in D, or will be concurrent with O and not yet in D. If there are any of the latter, then the D needs to be expanded to include them and O needs to be transformed to take account of their effect on the grove.

A motivating example Consider the authors Alice, Bob, and Carol collaborating on the document of Figure 6. Alice edits the text in node $(A, [0, 1, 1, 0, 1])$ of the grove, so her editor generates a $\mathrm{change}((\mathrm{A}, [0, 1, 1, 0, 1]), \text{"text"}, \ldots)$ operation and broadcasts it to Bob and Carol's sites. *At the same time*, Bob inserts the section of text rooted at $(B, [])$ into the main document as the second section of the chapter at $(A, [0, 1])$. His editor generates an $\mathrm{insert}((\mathrm{A}, [0, 1]), 1, \mathrm{B}, \mathrm{section})$ operation and broadcasts it to Alice and Carol's sites.

From Alice's point of view, the editor makes her change, then receives and honours Bob's change, and the net result *at her site* is illustrated in Figure 7. Alice's change (call the operation O_a) was applied to the place that Alice intended it to happen. The section of text that Bob wanted inserted is in the place Bob intended it to be (call this operation O_b). Alice sees both her and Bob's changes in a single consistent document. The history at Alice' site looks like $[O_a, O_b]$.

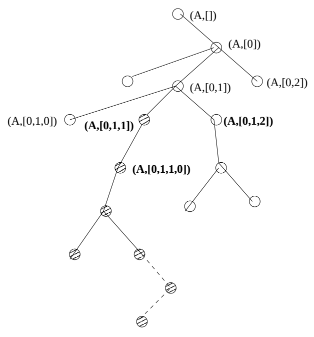

Figure 7: The end result at Alice' site.

From Bob's point of view, the editor makes his change then it receives Alice's change. Now, after node $(B, [])$ is inserted, the node at $(A, [0, 1, 1, 0, 1])$ is no longer the one that Alice was intending to change. Instead, $(A, [0, 1, 1, 0, 1])$ points to somewhere in the text that Bob has just pasted, and executing O_a immediately as is has two results:

1. The final state of the grove at Bob's site is not the same as Alice's in figure 7. There is no *convergence*.
2. The effect of Alice's operation O_a does not match its intended effect. There is no *intention preservation*.

The replicated grove would no longer be consistent, according to our consistency model. So Bob's editor should not execute Alice's operation as is. Instead, it should construct an operation $O_a{}'$ that points to the *right* node, and execute that. The problem is that O_a's original definition context does not include O_b (it hadn't happened at Alice's site at the time Alice generated O_a), but the execution context at Bob's site does include O_b. The solution is to transform O_a to have a definition context that includes O_a's original definition context, plus O_b. We write this $O_a{}' = \mathrm{IT}(\mathrm{O_a}, \mathrm{O_b})$, where IT is defined in the next section. We say Bob's editor transforms O_a against O_b. $O_a{}'$, the *execution form* of O_a at Bob's site, evaluates to $\mathrm{change}((\mathrm{A}, [0, 1, 1, 0, 1]), \text{"text"}, \ldots)$, and after executing $O_a{}'$, Bob's operation history is $[O_b, O_a{}']$ and Bob's grove looks the same as Alice's. We say that $[O_a, O_b]$ is *equivalent* to $[O_b, O_a{}']$, because both sequences of operations take the same initial state to the same final state.

From Carol's point of view, her editor receives O_b, applies Bob's edit to her local copy of the grove, and saves it in her

[4]If as each operation is executed it is stored in a history, the operations in the history form the current execution context.

history. O_a is then received from Alice's site, and her editor compares its vector timestamp with the operations in the history. It concludes that $O_a \| O_b$, applies IT to transform O_a in the same way as Bob's site, and saves that in its history. Carol then makes a change of her own, deleting the chapter at $(A, [0,1,0])$ in her grove. Her editor obeys, generates operation O_c $(\mathrm{delete}((\mathrm{A}, [0,1]), 0, \mathrm{C}))$, timestamps and broadcasts it to Alice and Bob. Meanwhile, Alice continues to edit the same text as before and her editor issues another operation O_{a1}, $\mathrm{change}((\mathrm{A}, [0,1,2,0,1]), \ldots)$, which is broadcast to Carol and Bob. As before, Bob's editor receives O_c. It checks O_c's timestamps against those of the operations in its history, and concludes $O_b \to O_c$, and $O_a{}' \to O_c$. O_c does not need transforming against these operations and is executed by Bob's editor as is. The operation history is now $[O_b, O_a{}', O_c]$. Then O_{a1} from Alice arrives. Again, the original definition context of O_{a1} does not match the execution context at Bob's site, because O_c had not reached Alice at the time O_{a1} was created. Bob's editor compares O_{a1}'s timestamps against its history, and finds that $O_c \| O_a 1$. To find an execution form with a current definition context equivalent to the current execution context of three operations, it computes $O_{a1}{}' \leftarrow \mathrm{IT}(\mathrm{O_{a1}}, \mathrm{O_c})$, where the inclusion transformation is defined in the next section.

Operational transformation is the technique of determining the difference between an operation's current definition context and the current execution context, and then rewriting the operation so that its current definition context is equivalent to the current execution context, so that when the transformed operation is executed, it will have the intended syntactic effect. Sun et al [19] defined the GOTO algorithm for this purpose. Given a causally-ready operation, and a history containing the current execution context, it returns a transformed version of the operation whose current definition context matches the history. GOTO proceeds as in the above example, comparing timestamps to those in the existing history to determine where definition and execution contexts do not match and applying transformation functions as necessary to make them match. For details, the reader is referred to [17].

The original application of GOTO was to the domain of flat linear text. That application comprised a data structure (an array of characters), an addressing scheme (array indexing), some operations on that data structure (insert and delete of strings), and definitions of IT and ET for those operations. In this paper we present a second concrete application of GOTO. In previous sections we have presented a data structure (the generalized grove), an addressing scheme (path vectors) and some operations (insert, delete, change). Below we present our definitions of IT and ET.

Inclusion Transformation

In this section we give our definition of the inclusion transformation function, discussing the first two cases using the example of the previous section. We will use the following standard notations for slices of lists: $N[a:b]$ means the a^{th} through b^{th} elements of a vector, inclusive. If a is omitted, it is assumed to be 0, if b is omitted, it is assumed to be $|N|$, the size of N. If y is [1,4,2], then y[:] is [1,4,2], and y[0:1] is [1,4]. A new vector with exactly one element x is written [x], just as [4,2,3] is a vector with three elements. The expression $[1,3,2] + [5,4]$ means to append $[5,4]$ to $[1,3,2]$, with a result of $[1,3,2,5,4]$.

Recall that Bob's editor had to transform Alice's first operation ($O_a = \mathrm{change}((\mathrm{A}, [0,1,1,0,1]), \text{"text"}, \ldots)$) against Bob's first operation ($O_b = \mathrm{insert}((\mathrm{A}, [0,1]), 1, \mathrm{B}, \mathrm{section})$). This was written as $O_a{}' = \mathrm{IT}(\mathrm{O_a}, \mathrm{O_b})$, where IT is defined:

```
$\mathrm{IT}(\mathrm{change}(\mathrm{N_a}, \mathrm{k}, \mathrm{f}), \mathrm{insert}(\mathrm{N_b}, \mathrm{n}, \mathrm{M}, \mathrm{T}))$
    $N_a{}' \leftarrow N_a$
    if $(\langle N_a \rangle = M)$
        $N_a{}' \leftarrow (\langle N_b \rangle, N_b[:] + [n] + N_a[:])$
    elif $(\mathrm{compare}(\mathrm{N_a}, \mathrm{N_b}) = \textsc{Prefix}(\mathrm{i}))$ and $(n \leq N_a[i])$
        $N_a{}'[i] \leftarrow N_a[i] + 1$
    return $\mathrm{change}(\mathrm{N_a}{}', \mathrm{k}, \mathrm{f})$
```

$O_a{}'$ is arrived at by comparing the grove addresses N_a and N_b for the two cases where O_b might affect the structure of the tree near where O_a operates. First, O_a's target might actually be on the branch that O_a inserts into the tree, as in Figure 8 If O_b has happened first, then that target node

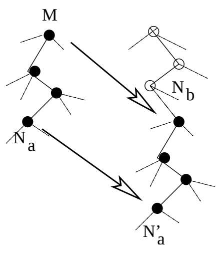

Figure 8: Inserting an existing branch.

is now a descendant of N_b and so $N_a{}'$ is prefixed by N_b. Second, if N_b is a Prefix of N_a, then O_b has inserted a new sibling next to the i^{th} node along N_a's path. For this example, the new sibling has pushed the target of O_a up one position, so $N_a{}'[i]$ is incremented to follow it, and $O_a{}' = \mathrm{change}((\mathrm{A}, [0,1,1,0,1]), \text{"text"}, \ldots)$.

Recall that Alice's second operation (O_{a1}) had to be transformed against Carol's first operation (O_c), written $O_{a1}{}' \leftarrow \mathrm{IT}(\mathrm{O_{a1}}, \mathrm{O_c})$, where $O_c = \mathrm{delete}((\mathrm{A}, [0,1]), 0, \mathrm{C})$. This case of IT is defined:

```
$\mathrm{IT}(\mathrm{change}(\mathrm{N_a}, \mathrm{k}, \mathrm{f}), \mathrm{delete}(\mathrm{N_b}, \mathrm{n}, \mathrm{M}))$
    $N_a{}' \leftarrow N_a$
    if $(\mathrm{compare}(\mathrm{N_a}, \mathrm{N_b}) = \textsc{Prefix}(\mathrm{i}))$
    if $(n < N_a[i])$
        $N_a{}'[i] \leftarrow N_a{}'[i] - 1$
    elif $(n = N_a[i])$
        $N_a{}' \leftarrow (M, N_a[i+1:])$
return $\mathrm{change}(N_a{}', k, f)$
```

As with the previous case of this transformation function, the targets N_a and N_b of the two operations are compared. If N_b is a PREFIX of N_a, then the deletion applies to a node on the path or one of its siblings. In this case, $n = 0$ and $N_a[2] = 2$, so a sibling of $(A, [0,1,2])$ is being removed, pulling that node to the left and leaving it with the address $(A, [0,1,1])$. That element of the path vector is decremented in the final execution form. If the deletion had taken out $(A, [0,1,2])$ (that is, if n had been equal to $N_a[2]$) instead, then we would have the situation of Figure 9. The victim's branch is now a new tree and a fresh path must be constructed from the pruned branch.

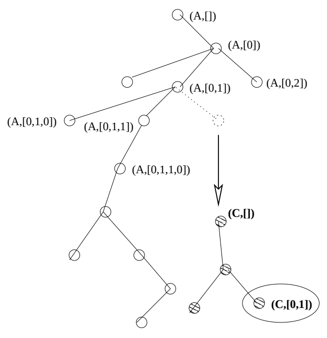

Figure 9: When a delete has moved a branch.

The other three cases for structural operations are given below without further discussion.

```
$\mathrm{IT}(\mathrm{insert}(N_a, n_a, M_a, T_a), \mathrm{insert}(N_b, n_b, M_b, T_b))$
    $N_a{}' \leftarrow N_a; n_a{}' \leftarrow n_a$
    if $(\langle N_a \rangle = M_b)$
        $N_a{}' \leftarrow (\langle N_b \rangle, N_b[:] + [n_b] + N_a[:])$
    elif $(\mathrm{compare}(N_a, N_b) = \mathrm{PREFIX}(i))$
        if $(n_b < N_a[i])$ or $(n_b = N_a[i]$ and $\mathrm{site}(N_a) < \mathrm{site}(N_b))$
            $N_a{}'[i] \leftarrow N_a[i] + 1$
    elif $(\mathrm{compare}(N_a, N_b) = \mathrm{SAME})$
        if $(n_b < n_a)$ or $(n_b = n_a$ and $\mathrm{site}(N_a) < \mathrm{site}(N_b))$
            $n_a{}' \leftarrow n_a + 1$
    return $\mathrm{insert}(N_a{}', n_a{}', M_a, T_a)$
```

```
$\mathrm{IT}(\mathrm{insert}(N_a, n_a, M_a, T), \mathrm{delete}(N_b, n_b, M_b))$
    $N_a{}' \leftarrow N_a; n_a{}' \leftarrow n_a$
    if $(\mathrm{compare}(N_a, N_b) = \mathrm{SAME})$
        if $(n_b < n_a)$
            $n_a{}' \leftarrow n_a{}' - 1$
    elif $(\mathrm{compare}(N_a, N_b) = \mathrm{PREFIX}(i))$
        if $(n_b < N_a[i])$
            $N_a{}'[i] \leftarrow N_a[i] - 1$
        elif $(n_b = N_a[i])$
            $N_a{}' \leftarrow (M_b, N_a[i+1:])$
    return $\mathrm{insert}(N_a{}', n_a{}', M_a, T)$
```

```
$\mathrm{IT}(\mathrm{delete}(N_a, n_a, M_a), \mathrm{insert}(N_b, n_b, M_b, T_b))$
    $N_a{}' \leftarrow N_a; n_a{}' \leftarrow n_a$
    if $(\mathrm{compare}(N_a, N_b) = \mathrm{PREFIX}(i))$
        if $(n < N_a[i])$ or $(n = N_a[i] \mathrm{and site}(N_a) < \mathrm{site}(N_b))$
            $N_a{}'[i] \leftarrow N_a[i] + 1$
    elif $(\mathrm{compare}(N_a, N_b) = \mathrm{SAME})$
        if $(n_b < n_a)$
            $n_a{}' \leftarrow n_a + 1$
    return $\mathrm{delete}(N_a{}', n_a{}', M_a)$
```

The case below has one interesting sub-case, where both operations are deleting exactly the same node, that is N_a is the SAME as N_b and $n_a = n_b$. In this case it should not be deleted a second time. We phrase this result as a change operation that makes no change at all (identity) to the children property in question.

```
$\mathrm{IT}(\mathrm{delete}(N_a, n_a, M_a), \mathrm{delete}(N_b, n_b, M_b))$
    $N_a{}' \leftarrow N_a; n_a{}' \leftarrow n_a$
    if $(\mathrm{compare}(N_a, N_b) = \mathrm{SAME})$
        if $(n_b < n_a)$
            $n_a{}' \leftarrow n_a - 1$
        elif $(n_b = n_a)$ and $(\mathrm{site}(N_a) = \mathrm{siteof} N_b)$
            return $\mathrm{change}(N_a, \mathrm{children}, \mathrm{identity})$
    elif $(\mathrm{compare}(N_a, N_b) = \mathrm{PREFIX}(i))$
        if $(n_b < N_a[i])$
            $N_a{}'[i] \leftarrow N_a[i] - 1$
        elif $(n_b = N_a[i])$
            $N_a{}' \leftarrow (M_b, N_a[i+1:])$
    return $\mathrm{delete}(N_a{}', n_a{}', M_a)$
```

Only structural operations can have an effect on grove addresses in operations they are transformed against, as changes to other properties do not affect the structure of the grove. Therefore,

```
$\mathrm{IT}(\mathrm{delete}(N_a, n, M), \mathrm{change}(N_b, k, f))$
    return $\mathrm{delete}(N_a, n, M)$
$\mathrm{IT}(\mathrm{insert}(N_a, n, M, T), \mathrm{change}(N_b, k, f))$
    return $\mathrm{insert}(N_a, n, M, T)$
```

Similarly, $\mathrm{IT}(\mathrm{change}(N_a, k_a, f_a), \mathrm{change}(N_b, k_b, f_b))$ returns the original operation unless the other operation has changed the *same* property on the *same* node ($N_a = N_b$ and $k_a = k_b$). Such a conflict is handled by defining transformation functions IT and ET to rewrite them to coexist in the same fashion as conflicting structural operations. Such have been defined by Sun and Chen in [19] for operations that insert a substring and delete a substring.

Exclusion Transformation

An exclusion transformation function is the dual to the inclusion transformation function, used to remove an operation from another operation's current definition context. The strategy is the same as for inclusion transformation - the grove addresses of the two operations are compared to see if one had affected the address of the target of the other, and appropriate adjustments made. Below is one case of $\mathrm{ET}()$.

```
$\mathrm{ET}(\mathrm{change}(N_a, k, f), \mathrm{delete}(N_b, n, M))$
    $N_a{}' \leftarrow N_a$
    if $(\langle N_a \rangle = M)$
        $N_a{}' \leftarrow (\langle N_b \rangle, N_b{}'[:] + [n] + N_a[:])$
    elif $(\mathrm{compare}(N_a, N_b) = \textsc{Prefix}(i))$ and $(n \le N_a[i])$
        $N_a{}'[i] \leftarrow N_a[i] + 1$
    return $\mathrm{change}(N_a{}', k, f)$
```

The local grove had O_a $(\mathrm{change}(N_a, k, f))$ and then O_b $(\mathrm{delete}(N_b, n, M)))$ applied to it. We attempt to find what O_b would have been had O_a *never happened*. If O_b had moved the branch containing O_a's target node $(\langle N_a \rangle = M)$ then had O_b never happened, the branch would be at N_b, and a path reflecting this is constructed. If O_b had removed a sibling of a node on O_a's path (N_b is a PREFIX of N_a) then that sibling would still be there if O_b had never happened, so the appropriate element of $N_a{}'$'s path vector is incremented. For each case of the inclusion transformation function, the corresponding case for the exclusion transformation function has been defined in like manner.

Related Work

Other work in Operational Transformation

The differences between this work and other editors that use operational transformation arise mostly from the domain (trees, not flat text) and the control algorithm used to apply the transformation functions. The GROVE editor [2] was the pioneering effort in this area. It transforms incoming operations against a linear history according to the dOPT algorithm, but counterexamples were given by Ressel et al [14]. They constructed a corrected version, the adOPTED algorithm, which requires an N-dimensional interaction graph as well as a linear history, and built an Emacs-like text editor with it. The REDUCE editor [19] uses the same GOTO algorithm as our current work, which only requires a linear history. We take its stringwise editing operations, and their transformation functions to handle changes to text-valued properties of grove nodes.

Other Editors of Trees

We are not aware of much work in group editing of trees; the projects we are aware of are discussed in this section. The MU3D editor [5] edits VRML and uses the VRML scene graph (a forest of trees) as its abstract data model. Like our work, it uses a path vector to address nodes in the tree and is fully replicated. In contrast to the unconstrained collaborative editing possible with operational transformation, MU3D maintains consistency by requiring users to lock a branch of the scene graph before modifying it. Other users may not work with a locked branch, limiting the options for collaboration, so conflicts are prevented rather than repaired.

Ionescu and Marsic [10] have taken an alternative approach to maintain consistency in the tree data model of XML. Their DISCIPLE application framework contains components that edit XML. As in this work, nodes in the tree are addressed by path vector from the tree root, and operations containing these addresses are broadcast by each site to its peers. Unlike this work, concurrent operations that interact are not rewritten to coexist, but instead are subject to arbitration according to their dARB algorithm. Operations whose sites lose arbitration rounds are annulled and their intentions lost. Our work enables these intentions to be preserved and visible to all authors, instead, and avoids the overhead of running their distributed arbitration algorithm at every conflict.

Ellis and Gibbs' GROVE editor [2] has a textual outline as its document type and the tree of outline entries as its abstract data model. As discussed, it uses operational transformation and so does not need locking or arbitration. Our work has the advantage of extending a standard data model (the grove) for widely-used document types (SGML, XML, HTML). Since the GROVE editor was developed before HTML, it used a data model and document type endemic to it, limiting its use as a tool for practical work.

Conclusions

This paper makes the following contributions to the fields of structured markup languages and computer-supported cooperative work:

1. An analysis of how the single-author data models in SGML hinder collaboration within the World Wide Web.
2. A scalable positional addressing scheme applicable to any grove representing any data structure, and a common set of fundamental operations for groves that is complete and parsimonious, as a basis for transformation functions.
3. A set of transformation functions for structural operations on a grove, suitable for use with the GOTO operational transformation control algorithm to remove this hindrance.

Before this work, the only method for synchronous collaborative editing of an XML document that did not require locking or other turn-taking techniques to maintain consistency was to use an operational-transformation text editor on the XML source. This fell far short of the useability of corresponding single-user tools for XML/SGML editing. Using our generalized grove and our transformation functions, existing single-user tools can readily be retrofitted to support collaborative work. These tools can be for XML/SGML, or any other document type with a grove-like abstract data model, such as VRML.

The next step is to finish building a reference implementation of the ideas in this paper. To this end, we are currently engaged in modifying Amaya, an open-source WYSIWYG editor written by the W3C's Document Formats Activity as a testbed for the XHTML (hypertext), SVG (vector graphics) and MathML (mathematics) dialects of XML. Internally it already uses a grove-like abstract data model, so the remaining work is to persuade instances of the editor to broadcast local operations and accept transformed remote operations for local execution.

After that, we will look at related techniques for handling the challenges of unconstrained synchronous concurrent editing. The GRACE editor [18] pioneered the technique of temporarily creating multiple versions of objects to accomodate conflicting, concurrent changes. We have already created an extension of the vanilla GOTO algorithm which uses multiple versioning as an option in cases where transformation is insufficient to preserve operation intentions. We will investigate the utility of this extended GOTO, and other techniques [15] [16] in the context of our grove work.

REFERENCES

1. Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, May 2001.

2. C.A.Ellis and S.J.Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 399–407. ACM, 1989.

3. Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. Technical report, World Wide Web Consortium, March 2001.

4. Ward Cunningham. Wiki Wiki Web. Web at http://c2.com/cgi/wiki?WikiWikiWeb, February 2002.

5. Ricardo Galli and Yuhua Luo. Mu3D: A Causal Consistency Protocol for a Collaborative VRML Editor. In *Proceedings of the Web3D-VRML 2000 fifth symposium on Virtual Reality Modeling Language*. ACM, February 2000.

6. Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. HTTP Extensions for Distributed Authoring - WEBDAV. RFC 2518.

7. ISO/IEC 8879:1986 Standard Generalized Markup Language (SGML), 1986.

8. ISO/IEC 10179:1996 Document Style Semantics and Specification Language, 1996.

9. ISO/IEC 10744:1997, Hypermedia/Time-based Structuring Language (HyTime) - 2nd edition, 1997.

10. Mihail Ionescu and Ivan Marsic. An arbitration scheme for concurrency control in distributed groupware. In *Proceedings of The Second International Workshop on Collaborative Editing Systems*. ACM, 2000.

11. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

12. David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the ACM 1995 Symposium on User Interface Software and Technologies*, pages 111–120. ACM, November 1995.

13. RELAX NG Specification. OASIS Committee Specification, http://www.oasis-open.org/committees/relax-ng/spec-20011203.html, 2001.

14. Matthais Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of ACM Conference on Computer Supported Cooperative Work*, pages 288–297. ACM, November 1996.

15. C. Sun. Undo any operation at any time in group editors. In *Proceedings of the ACM 2000 Conference on Computer supported cooperative work*, pages 191–200. ACM, December 2000.

16. C. Sun. Optional and responsive fine-grain locking in internet-based collaborative systems. *IEEE Transactions on Parallel and Distributed Systems*, page (to appear), 2002.

17. C. Sun and C.A.Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proceedings of ACM Conference on Computer Supported Cooperative Work*, pages 59–68. ACM, May 1998.

18. C. Sun and D. Chen. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Transactions on Computer-Human Interaction*, 9(1):63–108, March 2002.

19. C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, March 1998.

20. Document Object Model Level 1 Specification. W3C Recommendation, http://www.w3.org/TR/REC-DOM-Level-1/, 1998.