Generalizing Operational Transformation to the Standard General Markup Language

Aguido Horatio Davis, Chengzheng Sun, Junwei Lu,
Faculty of Engineering and Information Technology,
Griffith University, Queensland, Australia
horatio@qpsf.edu.au, {C.Sun, J.Lu}@mailbox.gu.edu.au

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

The World Wide Web exhibits a trend towards multiple authors interacting, in contrast to the traditional model of one author publishing to many readers. Research into collaborative editing is exhibiting a trend towards mainstream rich documents, away from idiosyncratic or functionality-poor document types suitable *only* for research. In this section, we argue that the intersection of these two trends is synchronous collaborative editing of instances of a structured, metadata-rich data model: trees.

The Web's original common case was, and is, one or a few authors generating operations on a document, and then publishing the result to many passive readers. The current abstract models for XML and HTML [21] and their parent language SGML [8] therefore are built from a single stream of events applied to a single mutable tree of objects. More recent initiatives such as Web Services [2] and the Semantic Web[1] leverage the Web as a medium for collaboration by revisiting these assumptions. More concretely, useful coarse-grained collaborative systems such as wikis [3] and Web Distributed Authoring and Versioning [6] tools are in production use now. These support coarse-grained collaboration, in that different authors can edit different HTML files concurrently. They do not yet 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 yet support synchronous collaboration, where each author is aware of the rest of the group's activity as it happens, because they have little notion of other instances of the document which should reflect remote changes. These limits hurt their flexibility and applicability.

To move beyond these limits, we must be able to collaborate *within* a document shared across multiple authors, and therefore need a data model that can maintain consistency in this environment. The state of the art in synchronous collaborative text editing, operational transformation, is useful exactly because it allows multiple authors to interact through a shared document with no additional constraints on their ability to do useful work. It allows the authors to act freely, and then reconciles those acts to yield a consistent document state, instead of maintaining consistency by forbidding certain freedoms to the authors. However, the major operational transformation editors [20] [15] [13] 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 the component shapes must be composable out of other shapes, and various levels of nesting must have data about shape, size, color, position associated with them. It cannot model SGML documents, because the elements of a document defined by tag pairs must be able to nest, and the elements need data from the tag properties associated with them. Therefore it cannot model documents written in the SGML dialects XML or HTML.

It can be seen that this data model and the operational transformation editors based upon it cannot deal with hierarchical, metadata-rich systems such as the World Wide Web except at the level of flat-text source code. To match the useability and power of existing single-user WYSIWYG tools, this constraint must be removed by defining techniques for operational transformation on a more general abstract data model. The discussion above points to trees of objects as a suitable basis for this model - perhaps not the only such basis, but as the one adopted by the ISO for transformation [9] and hypermedia [10] languages, by the W3C for XML [21] and various dialects [22] thereof, and by OASIS for schema languages [14], its track record as a basis for real world applications is good enough. In the next section we will propose such a model by adding concrete addressing and operations to a slight generalization of the SGML grove. In the third section we will discuss how to use operational transformation to apply multiple streams of operations to a replicated instance of our generalized grove. In the fourth section we will discuss related work on this problem, and in the fifth section we will present our conclusions and future work.

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

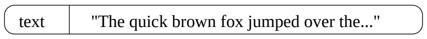

Figure 1: A paragraph as a single sequence of characters

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.

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

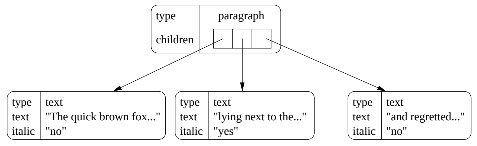

Figure 2: A paragraph as several sequences of characters

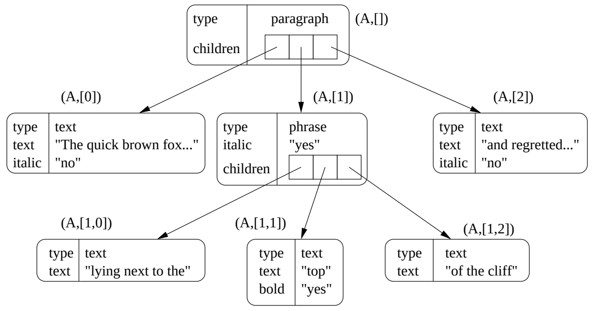

Figure 3: A paragraph as several sequences of characters

The data model this exemplifies is a tree of nodes. A node is a collection of properties, and has

- a type, which determines 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).
- zero or more ordinary properties which contain metadata

Our chosen 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 structure described above is essentially the grove defined in the Standard General Markup Language [8] and HyTime [10] standards as the abstract data model for SGML. The most important distinction is that SGML groves have exactly one subnode tree. Holding parts of the grove being worked on concurrently can require more than one; SGML content trees are also not replicated, and do not support real-valued numbers. Where the distinction matters, we will call our kind of grove a *generalized grove*.

In our architecture, each author’s editor holds a local replica of the shared document’s grove. Each operation is applied by the site whose author generated it as a *local operation*, and then broadcast to the other sites, who apply it as a *remote operation* on their groves to bring it back into consistency, possibly transforming it first.

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 *must have* the following characteristics:

1. Locatability. As well as reliably identifying nodes that exist, it is essential to be able to discuss locations that a node does not yet occupy, such as when placing a new node.
2. Consistency. The same address should refer to the same node across all sites which share an equivalent execution context.
3. Uniqueness. Each address must refer to at most one node in a given execution context; those which refer to exactly one are valid, those which refer to a location with no node in it are invalid.

All else being equal, we *prefer* lighter weight (addresses which occupy less space and algorithms which are less complex and costly) domain independent (robust with respect to notation or kind of grove) and completely distributed (dereferencing or generating an address does not require message exchange which costs time, bandwidth and complexity) addressing schemes.

To fulfill these requirements, we can name all nodes and address each by name, or choose certain nodes as landmarks and address each other node by its 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[1] - but in the broader context, naming *everything* has two problems.

First, a location in the grove 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 an author

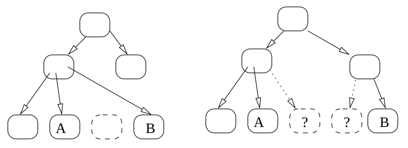

Figure 4: Why pure naming is ambiguous

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[2] this *landmark problem* will recur.

Second, non-trivial hypertexts such as wikis [3] 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 intrinsic to the names, a table must be kept mapping names to addresses. This costs one record of space per named node, which scales poorly to large documents with large numbers of nodes.

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; we write $|N|$ for its cardinality -

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

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

$|[1,3,2]| = 3$. Alternately, a complete grove address is written $(R, [i_1, i_2, \dots])$, with $|(R, N)|$ meaning $|N|$. With such addresses, the mapping between an anonymous node and its address is stored in the grove structure, instead of bulking up a lookup table. We choose to give names only to the root of the subnode tree(s), which cannot be described as child of another node. Figure 3 shows the grove with the addresses next to each node as a concrete example.

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 $\mathrm{insert}(\mathrm{N}, \mathrm{n}, \mathrm{M}, \mathrm{T})$ 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 fresh node is created and inserted as the n^{th} child of N. In either case, the node has type T.

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. When the shaded node is deleted, its descendants must

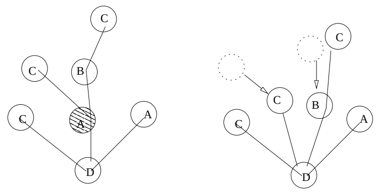

Figure 5: Why deleting single nodes is undesirable

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 yet allowing others, 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 save the excised branch as another subnode tree whose root is labelled with the M operand of the delete - such a tree can be garbage collected when remote operations from all sites have arrived that are causally after the deletion. Figure 9 is an example of this - $\mathrm{delete}((\mathrm{A}, [0,1]), 2, \mathrm{C})$ does not *destroy* $(A, [0,1,2])$, it severs it from the tree and renames it to $(C, [\,])$. 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 $\mathrm{change}(\mathrm{N}, \mathrm{k}, \mathrm{f}, \dots)$ will change the content data or metadata of a grove node whose address is N. Before it is executed, the property with key k has value v. After the change is executed, the property has the value $f(v)$. The transition function f, which takes the old value and returns the new value, can in principle be arbitrary. For the special case of SGML, HTML, or XML, the relevant transitions are text insertion and deletion; "insert 'ar' at position 35 in the text", "delete the 5 characters starting at position 28 in the `author` property". Note the phrasing as a change in the existing value, one which requires other parameters to completely define it; these are included as the fourth and succeeding operands of the change operation. [20] describes the semantics of stringwise insert and delete in more detail, and defines inclusion and exclusion transformations for this special case.

Strictly speaking, the minimal complete set of operations is node creation and deletion. Mutating a node *could* be expressed as deleting that node and creating 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. More seriously, it is ambiguous. A concurrent remote operation on a node which has already been deleted (pruned) locally would be lost, as there is no way to know the old and new addresses refer to the same node, and the old address would point either to a location with no node in it, or to the wrong node.

STRUCTURAL OPERATIONAL TRANSFORMATION

Background

A consistency model We adopt a consistency model from our previous work in collaborative editing [20] [18]. 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 each grove are isomorphic 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. In line with our previous approach, we define the intention of O as the syntactic effect of O at the site of its origin.

We enforce causality by the method used in [4]; maintaining a vector clock at each site and timestamping each operation with the current value of the local clock before broadcast. Operations whose timestamps indicate they are not *causally ready* for execution are delayed until they are. It can be shown that local operations are 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.

2. The first and second paths point to the SAME node. An operation at N_b could not affect the address of the node at N_a, only the addresses of N_a's children.

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 might 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 would 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.

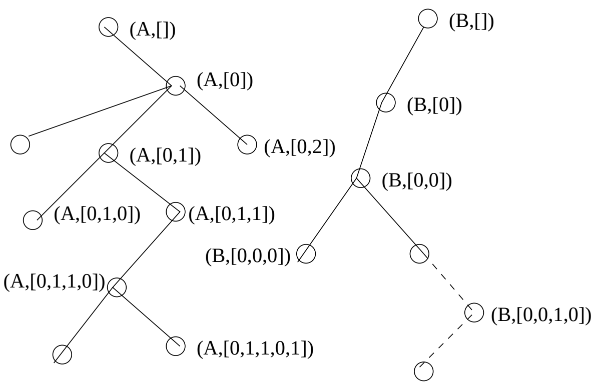

Figure 6: Comparing paths in an example grove

Context To discuss preserving operation intentions, we must introduce the notion of *operation context* [19]. 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 that informed the original author when they issued the operation. 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^3. 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 oper-

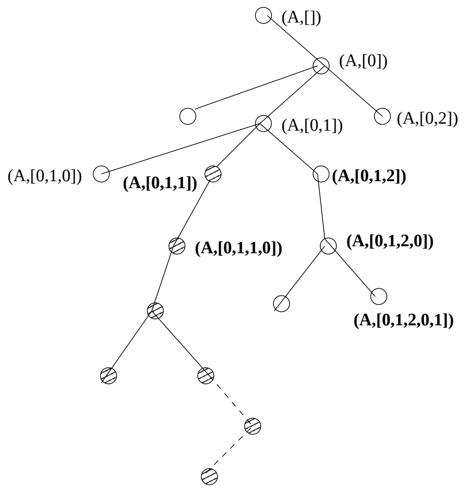

Figure 7: The end result at Alice' site.

ation 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 is now $[O_a, O_b]$.

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

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 $O'_a \leftarrow \mathrm{IT}(\mathrm{O_a}, \mathrm{O_b})$, 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, is $\mathrm{change}((\mathrm{A}, [0, 1, 1, 0, 1]), \text{“text”}, \ldots)$, 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 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 did, executes the result, 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 $O_b \rightarrow O_c$, and $O'_a \rightarrow O_c$. O_c does not need transforming against these operations and is executed by Bob's editor as is. His 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 $O_c \| O_{a1}$. To find an execution form with a current definition context equivalent to the current execution context of three operations, it computes $O'_{a1} \leftarrow \mathrm{IT}(\mathrm{O_{a1}}, \mathrm{O_c})$, using the inclusion transformation defined in the next section.

Operational transformation is this 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. In [20] we 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 [19].

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 $|N| - 1$. 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 ($O_a = \mathrm{change}((\mathrm{A}, [0,1,1,0,1]), \text{“text”}, \dots)$) against Bob's first operation ($O_b = \mathrm{insert}((\mathrm{A}, [0,1]), 1, \mathrm{B}, \text{section})$). This was written as $O'_a = \mathrm{IT}(\mathrm{O_a}, \mathrm{O_b})$. This case of 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 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 $O'_a = \mathrm{change}((\mathrm{A}, [0,1,1,0,1]), \text{“text”}, \dots)$.

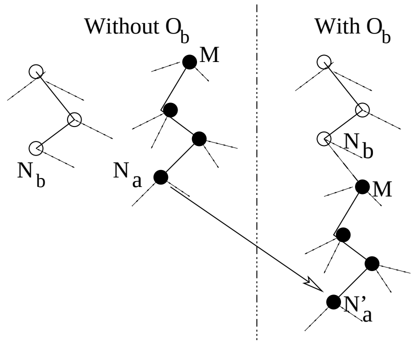

Figure 8: Inserting an existing branch.

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 $\mathrm{change}(\mathrm{N'_a}, \mathrm{k}, \mathrm{f})$
```

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 would now be a new tree and a fresh path would need to be constructed from the pruned branch.

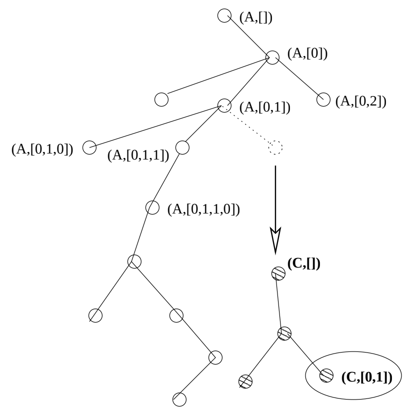

Figure 9: When a delete has moved a branch.

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

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

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

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

The next 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 ($\mathrm{identity}$ transition function) to the children property in question.

```
$\mathrm{IT(delete(N_a, n_a, M_a), delete(N_b, n_b, M_b))}$
    $N'_a \leftarrow N_a; n'_a \leftarrow n_a$
    if $(\mathrm{compare(N_a, N_b) = SAME})$
        if $(n_b < n_a)$
            $n'_a \leftarrow n_a - 1$
        elif $(n_b = n_a)$ and $(\mathrm{site(N_a) = siteofN_b})$
            return $\mathrm{change(N_a, children, identity)}$
    elif $(\mathrm{compare(N_a, N_b) = PREFIX(i)})$
        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 $\mathrm{changes}$ to other properties do not affect the structure of the grove. Therefore,

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

Similarly, $\mathrm{IT(change(N_a, k_a, f_a), change(N_b, k_b, f_b))}$ 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. For the special case of string mutation (inserting a substring or deleting a substring), such have been previously defined in [20].

Exclusion transformation

Transformation control algorithms such as GOT [20] and GOTO [19] save each operation O_x in the history, not as it was received from the original site, but as it was executed locally after being transformed. If a remote operation O_y arrives whose original definition context contains a differently transformed form of O_x it may be necessary to recover the original O_x. So, there is a need for a dual to IT, the exclusion transformation ET such that $\mathrm{ET(IT(O_a, O_b), O_b) = O_a}$. The strategy is very similar to an inclusion transformation - compare the grove addresses of the two operations to see if O_b had affected the address of the target of O_a. If so, reverse whatever the effect of O_b on O_a was and return an O_a

adjusted to what it would have been had O_a operation never happened. As an example, 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 $(n \leq N_a[i])$
        $N'_a[i] \leftarrow N_a[i] + 1$
    return $\mathrm{change}(N'_a, k, f)$
```

Here, O_a $(\mathrm{change}(N_a, k, f))$ and then O_b $(\mathrm{delete}(N_b, n, M)))$ had been applied to the local grove. We attempt to find what O_b would have been had O_a *not happened before it*. 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

Operational transformation originated with Ellis et al [4] and was further developed for synchronous collaborative editing of flat text [15] [20] [13]. The differences between this previous work and our current work are principally the domain (hierarchical trees, not flat text) and/or the control algorithm used to apply the transformation functions (exclusion as well as inclusion transformation).

Other Editors of Trees

To the best of our knowledge, there has not been much work in group editing of trees; what exists is 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 [7] 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 our 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 instead enables these intentions to be preserved and visible to all authors, incidentally avoiding the overhead of running a distributed arbitration algorithm at every conflict.

Molli et al [12] treat a tree of nodes as a single shared object, opt to name each node (in contrast to our approach of landmarks), and only use operational transformation for mutations, not structural operations. Their XML/HTML editor adopts the same consistency model [20] as our work, but the transformation functions they give seem not to preserve syntactic intentions as we define them. Their nodes are not derived from the existing SGML standards.

CONCLUSIONS

This paper contributes a set of transformation functions for structural operations on a grove, that used with the GOTO operational transformation control algorithm [19] and a set of transformations for mutation operations such as [20] will enable synchronous collaborative editing of any metadata-rich hierarchical content. In support of these, we have defined 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.

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 falls far short of the useability of existing WYSIWYG single-user tools for XML/SGML editing. Existing single-user tools can readily be retrofitted to support collaborative work by exposing their state as a generalized grove and melding streams of events on that grove from all users with our transformation functions. This state can be documents written in SGML dialects, data- and knowledge bases stored in XML dialects, vector graphics represented as hierarchies of visual objects, or worlds written as VRML or X3D scene graphs. Indeed, this technique seems suitable for any work that can be expressed as a series of operations applied to an instance of a content type with a grove-like abstract data model.

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 creating multiple versions of objects to accomodate conflicting, concurrent changes. We are looking at 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 [17] [16] in the context of our grove work.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments, which were invaluable for improving the presentation. This work was supported by Australian Research Council Large Grant A00000711.

REFERENCES

1. Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, pages 13–21, May 2001.
2. Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. Technical report, World Wide Web Consortium, March 2001.
3. Ward Cunningham. Wiki Wiki Web. Web at http://c2.com/cgi/wiki?WikiWikiWeb, February 2002.
4. Clarence 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, May 1989.
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*, pages 53–62. ACM, February 2000.
6. Yaron Goland, James Whitehead, Asad Faizi, Steve Carter, and Del Jensen. HTTP Extensions for Distributed Authoring - WEBDAV. RFC 2518.
7. Mihail Ionescu and Ivan Marsic. An arbitration scheme for concurrency control in distributed groupware. In *Proceedings of The Third International Workshop on Collaborative Editing Systems*, pages 32–42. ACM, 2000.
8. ISO/IEC 8879:1986 Standard Generalized Markup Language (SGML), 1986.
9. ISO/IEC 10179:1996 Document Style Semantics and Specification Language, 1996.
10. ISO/IEC 10744:1997, Hypermedia/Time-based Structuring Language (HyTime), 1997. Second edition.
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. Pascal Molli, Hala Skaf-Molli, Gerald Oster, and Sebastien Jourdain. SAMS: Synchronous, asynchronous, multi-synchronous environments. In *Proceedings of the Seventh International Conference on CSCW in Design*. International Working Group on CSCW in Design, September 2002. (to appear).
13. 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.
14. RELAX NG Specification. OASIS Committee Specification, http://www.oasis-open.org/committees/relax-ng/spec-20011203.html, 2001.
15. 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.
16. Chengzheng Sun. Optional and responsive fine-grain locking in internet-based collaborative systems. *IEEE Transactions on Parallel and Distributed Systems*, 13(9):994–1008, September 2002.
17. Chengzheng Sun. Undo as concurrent inverse in group editors. *ACM Transactions on Computer-Human Interaction*, 10(1), March 2003. (to appear).
18. Chengzheng Sun and David Chen. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Transactions on Computer-Human Interaction*, 9(1):1–41, March 2002.
19. Chengzheng Sun and Clarence 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.
20. Chengzheng Sun, Xiaohua Jia, Yanchung Zhang, Yun Yang, and David 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.
21. Document Object Model Level 1 Specification. W3C Recommendation, http://www.w3.org/TR/REC-DOM-Level-1/, 1998.
22. Scalable Vector Graphics 1.0 Specification. W3C Recommendation, http://www.w3.org/TR/SVG/, 2001.