Collaborative Editing of XML Documents - An Operational Transformation Approach

A.H.Davis, C.Sun and J.Lu
Faculty of Engineering and Information Technology
Griffith University
Nathan, Queensland 4111 Australia
horatio@qpsf.edu.au, scz@cit.gu.edu.au, j.lu@me.gu.edu.au

ABSTRACT

Collaborative editing embraces many kinds of document, with as many different, domain-specific solutions. We hypothesize that adopting XML as a unified representation of many of these editing domains and creating a single consistency-maintenance algorithm for XML's abstract data model will yield a general solution to the synchronous collaborative editing problem over all of them. It is well-understood that an XML document's data model is a planar, rooted, decorated tree. This paper presents initial work on testing our hypothesis by adapting the Generic Operational Transformation (Optimized) algorithm to apply to such trees of document elements instead of linear arrays of characters.

Keywords

Groves, concurrent update, synchronous collaborative editing, XML, SGML, Amaya, operational transform.

INTRODUCTION AND MOTIVATION

Why synchronous collaboration?

We seek to use a collection of documents as a medium for fluid interaction between (and amongst) its authors.[1] The kind of work to be supported is closely cooperative, such as exploratory problem analysis or brainstorming or critical editing of a document's contents or discussion of simulation results by a group. The primary goal is an unstructured, unplanned flow of ideas and reactions between the authors, as expressed in the changing contents of the documents (that is, by collaborative editing).

Asynchronous modes of collaboration, by definition, do not support such immediacy well. Consider the distributed independent and the centralized turn-taking approaches. If the collaborators separately update local copies of the documents involved, group awareness is coarser-grained, cooperation is less fluid, and costs for post-hoc reconciliation of the replicas are incurred. If the collaborators share a copy of the documents but must use floor-control mechanisms to ensure document consistency, group awareness is possible and reconciliation is unnecessary. However, cooperation between different users cannot be immediate or fluid because only one (the floor holder) is free to act. In either case, asynchronous collaborative editing is not appropriate, so we must solve the harder synchronous collaborative editing problem.

[1]We use "author" in the general sense of "author or reader". All members of the collaboration should be free to contribute to the document, but will not want to exercise that freedom all of the time.

Why a universal notation for content?

Both logistical and social problems arise when all authors must agree to use a particular software tool for their group work. A site may not have the tool available, or the user may not have the privileges to install it, or the site may not have enough computational resources to support it (an issue in mobile computing). The user may have a tool or environment they prefer to use for the task, because it is more capable or faster or better presented than the unfamiliar collaborative equivalent. We argue therefore that synchronous collaborative editing should be implemented by finding and exploiting common ground between all these existing tools, rather than replacing them with purpose-built collaboration-aware tools.

The document notation, the underlying abstract data model and set of fundamental operations form a suitable common ground. Two users may disagree over whether `vi` or `emacs` is the best editor, but they will both agree that they are editing ASCII text and their abstract data model is a linear buffer of characters which they add, delete, and move. The discourse between cooperating sites can then be in terms of operations applied by each site to a common instance of that abstract data model. Which tool each user uses to apply those operations can be suited to the user, the user's role, and the site, and existing tools can be leveraged by adding code to let them speak to other sites in terms of operations.

We argue that a single notation, data model and operations should be chosen that are general enough to represent all the document types of interest. For example, a document used in concurrent engineering would ideally support rich text, two- and three-dimensional graphical visualization, mathematical notation, and code, because these content types are all of interest to engineers who use computers as tools for their work. If each content type has a separate notation (and abstract data model, and operations) then a different set of consistency maintenance techniques must be devised for each. If a useful working set of content types can be expressed in terms of a single, unified notation (and abstract data model, and operations) then only a single solution for synchronous collaborative editing of that notation is needed.

The Extensible Markup Language is a good candidate for this unified notation, because it was specifically designed as a universal markup language. Dialects of XML exist and are in common use for rich hyperlinked text [12], vector graphics [14], mathematical notation [13], chemistry [5], ontologies [6], multimedia presentations [15], resource metadata [11] and topic maps [10]. There is a rich set of general tools for parsing, transforming, rendering, and working with XML. There is also a growing set of tools in each application domain - often XML is used as a lingua franca between originally incompatible software and databases. Collaborative tools that operate on XML leverage all this existing infrastructure to provide a capable and *interoperable* environment for collaborative work. This gives users active incentives to switch to CSCW.

XML and the Grove Abstraction

Our goal is to unify the abstract data model (by choosing a unified document notation) and then solve the synchronous collaborative editing problem for this data model. For the reasons given above, we choose XML as this representation. XML is an application of the Standard General Markup Language and therefore we choose the formal data model of SGML as our data model. This data model is the grove, as defined in the relevant SGML [3] and HyTime [4] standards, which is the well-understood tree model for a hierarchical document.

A grove is a set of nodes connected by edges, organized into one or more trees. These *content trees* are built from directed edges that model a "contains" relation between elements of a document. Each node models an element of the document structure as an ordered collection of assignments between property keys and property values ("`font`" = "`sans-serif`", for example). These encode both content data (the actual content of the document) and metadata (attributes such as font, size, origin, hyperlinks). For each kind (class) of node, one property is distinguished as the content property. If the node is a leaf of a content tree, it holds document content (such as a string of plain text) in its content property directly. If the node is an internal node of a content tree, it has children, which model child elements of the document (for example, a node for a HTML `<UL>` tag pair has one child for each list item). Such a node's content property is a list of child nodes. A content tree has a root (for example, the node for a `<HTML>` tag), and if the grove contains more than one tree, the grove has a root. Other edges may be present denoting arbitrary relationships between nodes. Groves are constructed from some notation (e.g. HTML) according to a property set by a notation processor. Property sets are schemas which govern the permitted classes, properties, and roles of nodes in a grove.

As a concrete example, consider the quasi-XHTML document in figure 1 and a corresponding grove, shown in figure 2. The grove has one tree, which completely models each element in the document. The node modelling `<HTML></HTML>` has as children all the nodes modelling child elements. It, in turn, is the child of a node that models the whole document. The visible leaf nodes model pieces of plain, unattributed text. For simplicity, the metadata attached to some nodes (priorities, user-assigned labels) is only partially shown.

The formal definition of a grove is abstract enough to be easily reasoned with, but concrete enough to be easily mapped to an implementation (e.g. the DOM) if two additional pieces of information are defined: a set of fundamental operations on groves, and an addressing scheme to reproducibly refer to a node within a grove. We have chosen definitions for these appropriate for work with SGML-based notations. These definitions and some supporting discussion are presented in the next section.

System Architecture

This section briefly reviews the relevant parts of the REDUCE collaborative editing framework and indicates how the present work fits into that framework. A detailed treatment of, and motivation for this framework can be found in [9].

Each site has a local copy of the grove representing the document, and executes the local users' editing operations on that copy immediately. The operations are then timestamped and broadcast to the other sites, which execute them on their local copies in the appropriate order. In general, the definition context (state of the grove which informs the user who issued the operation) and the execution context (state of the grove on which the operation will be executed) are only the same at an operation's origin. The execution context elsewhere may differ by the effect of several concurrent operations. A modified form of the original operation must be found that preserves the intended effect in the face of this. A good and complete solution to the problem of finding such a transformed operation is the Generic Operation Transformation (Optimized) algorithm, GOTO [8].

GOTO is generic across any data model, and any set of operations on that data model. The algorithm deduces the causal relationship between the current execution context and the definition context of an incoming operation from its vector timestamp, and then applies some transformation functions to make the definition and execution contexts match of that operation match. These transformation functions encapsulate the detailed semantics of a given set of operations for a given abstract data model. An example set of transformation functions for a linear array of immutable characters which supports `insert` and `delete` operations can be found in [9].

The focus of the present work is to apply GOTO to groves by writing a set of transformation functions for them. Differences in addressing between a linear array of characters and a hierarchical tree of document elements make this a nontrivial exercise. This would contribute a general solution to collaborative editing of *anything* which can be modelled as a grove. Unlike previous solutions for collections of trees in the domains of VRML [1] or XML [2], this solution preserves intention without requiring locking or arbitration. It is a complete solution. The finite amount of complexity involved in getting a complete and correct solution for groves could then be reused for *all* of these domains, and more.

GOTO AND GROVES - SOME CONSIDERATIONS

In this section, we present a grove addressing scheme, a set of fundamental operations that take these grove addresses as operands, and sketch some considerations in constructing a set of transformation functions using both.

Addressing

We seek an addressing scheme for groves with the following characteristics:

1. Lightweight addresses. The basic transformation algorithm requires relatively many comparisons between addresses, so they need to be cheap to compare. Consistency maintenance requires transmitting and storing the addresses which are operands, so they need to be cheap to represent.

2. Globally consistent addresses. The same address should point to the same node for each replica of the grove, else an additional mapping between addresses at different sites would need to be stored to translate the addresses of foreign operations.

3. Independence from particular property sets or document notations. The goal is a general solution, and relying on assumptions peculiar to one notation or one kind of grove would defeat that generality.

4. Addressability of unpopulated locations in the tree. The address where a newly-created node should be deposited is an example of this.

5. Completely distributed operation. Dereferencing or generating an address should not require message exchange between sites. Message exchange costs extra time, and could make the system vulnerable to race conditions.

A straightforward approach is to label each node independently of its place in the grove, and then address all nodes by label. This approach can (with suitable choice of labels) be globally consistent, notation-independent, and completely distributed. However, it incurs costs for storage of a label with each node, and storage for the mapping from labels to the nodes. It also requires an additional algorithm to generate unique, valid, and globally consistent labels.

Clearly we should label as few nodes as necessary, and exploit the regular structure of the data model to identify the other nodes. Because the children of a node are ordered, a lightweight address for a non-root node can be constructed from its position in a tree. An example of such a positional address is a C array; it is typically addressed by a pointer to the base element, plus an offset from the base to the element of interest. The actual offset never has to be stored with each element. It is the same here. As there is no natural order between trees in a grove,[2] our addressing scheme labels the root nodes of content trees. All other nodes will have a positional address composed of the relevant root, plus a description of the path from the root to the node.

Unlike arrays, a tree has a hierarchical structure which requires one index for each level of the tree along the path. Each index is the number of siblings to the left of whichever node is being discussed - the first child of a parent has sibling index 0, the tenth child has sibling index 9. A path is then written as a (possibly empty) vector of these indices, with those nearest the root first. The empty path vector `()` denotes the root of the tree. In the example of figure 2, this is the document node at the bottom of the diagram. The path vector `(0,1)` denotes the second child of the first child of the root (indexing start from 0). In the example, this is the node representing the second ordered list ("Recommendations"). The path vector `(1,0,3,5)` denotes the sixth child of the fourth child of the first child of the second child of the root. In the example, this location is buried somewhere in the nodes representing the SGML DTD of the document. The path vector `(0,0,1,2)` denotes the third child of the second child of the first child of the first child of the root, which models the phrase "be avoided" in the second point of the conclusions. The complete address for this is written `{A,(0,0,1,2)}`, if 'A' is the label for the content tree. It is sometimes useful to have a null value for an address; this null address is written `None`.

For each tree root, there exists a unique operation that caused it to become a tree root. As all operations are uniquely timestamped, a suitable label for such nodes can be extracted from the timestamp of the creating operation.[3] For simplicity, we will label the tree(s) in our examples 'A', 'B', etcetera in order of creation.

Fundamental Operations

The REDUCE approach specifies higher-level editing operations as sequences of fundamental operations. We have chosen a set of fundamental operations to suit groves, according to the criteria:

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

2. *Soundness* - applying any sequence of fundamental operations to a valid grove yields a valid grove.

3. *Disambiguity* - each fundamental operation (and its operands) contains enough information to support correct, consistent do and undo[7].

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

[2] One could be constructed from the order in which they are attached to the grove root, but this would not reflect the document structure in a useful way.

[3] A heuristic for making suitable labels is to compose a tuple of the site of origin n and the n^{th} component of the vector timestamp.

Using these criteria, we have chosen the fundamental operations `add`, `delete`, `move` and `change` for groves. The first three are *structural* operations - they make changes to the structure of the grove. The latter is a *mutation* operation - it changes the contents of a node without changing the structure of the grove.

The `add` operation creates a node at a given location in the grove. It takes three operands. The first is a grove address (tree identifier plus path) pointing to the location in the grove where the node is to be created. The second is a string identifying the class of which the new node will be an instance. The third is an initial set of properties for the node. Figure 3 shows an example - the operation $\texttt{add}(\{\texttt{A}, (0, 1)\}, \ldots)$. In particular, note the effect on the addresses of the new node's siblings. What used to be at $(0, 1)$ is now at address $(0, 2)$. All the paths that used to start $(0, 1 \ldots)$ now start $(0, 2 \ldots)$.

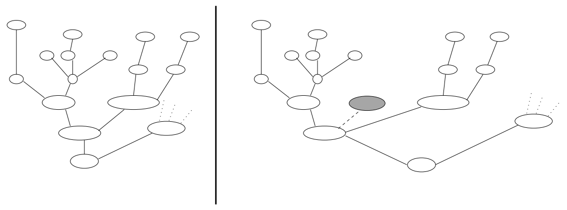

Figure 3: Example of `add` operation

The `delete` operation deletes a node from a given location in a grove. It takes one operand, the grove address of the node to be deleted. The deleted node and its descendents (if any) form a temporary tree which is flagged for garbage collection when safe. Such dead branches may be safely destroyed after all concurrent operations on that branch (if any) have executed, and no operation in the execution history refers to that branch.

An alternative way to implement deletion is to replace the deleted node with its children instead of removing the entire branch. However, the property set in force may forbid the parent-child relationship between nodes of certain classes (a `<P>` node should not be the parent of other `<P>` nodes in HTML). So, the rules would have to change on a per-application basis. A consistent rule that children follow their parent is preferable.

Figure 4 illustrates the operation $\texttt{delete}(\{\texttt{A}, (0, 0)\})$. Afterwards, there are two trees. The shaded tree root is flagged for garbage collection. Again, note the effect on the addresses. All paths that start with $(0, 1 \ldots)$ now start with $(0, 0 \ldots)$.

The `move` operation changes the structure of the grove by moving an existing node from one location within the grove to another. It is an atomic composition of the `delete` and `add` operations, and thus takes two operands. The first operand is the grove address of the node to be moved. The second is the grove address of the location to which the node is to be moved. As with `delete`, the descendents of the target node follow it to the new location, for similar reasons. Figure 5 illustrates the operation $\texttt{move}(\{\texttt{A}, (0, 1)\}, \{\texttt{A}, (0, 0, 1)\})$.

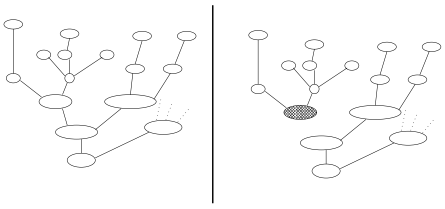

Figure 4: Example of `delete` operation

As `move` is composed of `delete` and `add`, its effect on addressing can potentially be that of a `delete` and an `add`. Figure 5 shows the latter.

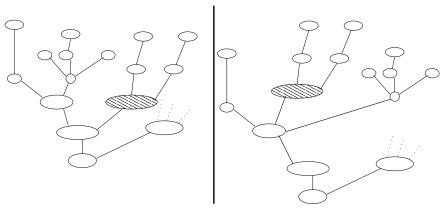

Figure 5: Example of `move` operation

The `change` operation updates a property on the node at a given location in the grove. It takes three operands. The first is the grove address of the node to which the property belongs. The second operand is the property's key. The third operand specifies an update function. On execution, the value of the appropriate property on that node is looked up and the update function is applied to it. The result is then the new value of that property. In principle the update function can be arbitrary. The update functions we are focussing on in the present work are:

- Complete update. The update function returns a fixed new value irrespective of the old value.
- Stringwise insertion and deletion. The update function incrementally edits a string property, such as text content.

Strictly speaking, the minimal complete and sound set of operations is `add` and `delete`. The `change` operation may be expressed as a `delete` on the relevant node and an `add` to create a node with the changed property in the same location. The `move` operation may similarly be expressed as a `delete` of the node in the old location and a series of `add`s to reconstruct the branch in the new location. However, such a minimal operation set would be neither disambiguous nor parsimonious.

For a `move`, the other collaborating sites would receive a string of operations which reconstruct the state of the relevant branch *as seen by the originating site.* This only results in a consistent global state if the other collaborating sites have not made concurrent local changes to their copies of the branch. This is because the reconstructed branch will not contain the changes made. If the movement operations and the concurrent changes are interleaved on arrival at other sites, the results will depend on the exact sequence of arrival, because GOTO will not have enough information to rewrite the addresses to point to the new branch. A separate, atomic `move` operation will encode the missing information: that the nodes in the old branch and the nodes in the new branch are the same nodes. This is why our current set of operations includes one. A `change` expressed as `adds` and `deletes` would reconstruct the branch in place, and thus would fall prey to the same race condition. That is why `change` is included as a separate operation. Both of these design decisions incidentally support parsimony by replacing a deletion and n additions by a single operation, for an affected node with n descendents.

Constructing an Inclusion Transformation Function

Under GOTO, an inclusion transformation function is applied to append an operation to another operation's definition context; the function `include(`O_x, O_y`)` yields a new operation $O_x{}'$ which includes O_y in its definition context but otherwise has exactly the same effect as O_x. In this subsection we will consider the case where both O_x and O_y are structural operations, and sketch the derivation of an algorithm to compute this inclusion transformation.

We analyze each (`add`, `delete`, or `move`) as a generic sequence of events. First, a node is deleted from a *source address* in the grove. Second, that node is added at a *destination address* in the grove.[4] The exact mapping from the concrete operations (`add`, `delete`, `move`) to a generic structural operation is dependent upon the operation. For a `move`, the source and destination addresses map naturally to the first and second operands of the operation; in the operation of figure 5 the source address is `{A,(0,1)}` and the destination address is `{A,(0,0,1)}`. For an `add`, the destination address is the first operand (the location where the node will be added), but the source address is a null address; the node "comes from" nowhere. Conversely, for a `delete`, the source address is the first and only operand because that is where a node will be removed from. In the long term, the node is not replaced anywhere, so the destination address is a null address[5].

From this viewpoint, the operation O_x, operating on a grove which matches its definition context, would therefore achieve the following effect:

1. Remove a node N from the grove at address s_x.
2. Replace the node N in the grove at the address d_x.

where $s_x = \text{source}(O_x)$, $d_x = \text{destination}(O_x)$, The sequence of events that includes O_y in O_x's definition context would be:

[4]As previously stated, this notional removal and replacement affects that entire branch.

[5]There are a couple of corner cases in the transformation where O_x addresses a node in the branch that O_y has deleted; in these cases the destination address of O_y is the root of the temporary tree formed from the dead branch. This is outside the scope of this paper.

1. Remove the node M from the grove at address s_y.
2. Replace the node M in the grove at the address d_y.
3. Remove the node N from the grove at address $s_x{}'$.
4. Replace the node N in the grove at the address $d_x{}'$.

where $s_y = \text{source}(O_y)$, $d_y = \text{destination}(O_y)$, $s_x{}' = \text{source}(O_x{}')$, and $d_x{}' = \text{destination}(O_x{}')$.

We proceed by deriving the source and destination addresses $s_x{}'$, $d_x{}'$ of the new operation $O_x{}'$ using these two sequences of events. To derive $s_x{}'$ we

1. Set $s_x{}' = s_x$.
2. Correct $s_x{}'$ for the effects of M being removed from the grove at address s_y. If this node was removed from the left of or below N, N's position in the tree has changed, so the positional address - $s_x{}'$ - must change to point to where it now is.
3. We next correct $s_x{}'$ for the effect on addressing of M being added to the grove at address d_y. Again, if O_y added this node to the left of or below where N will end up, that position in the tree will change (see the discussion of figure 3) and the address must change also.

The resulting source address for $O_x{}'$ now has all the events of O_y in its definition context - it is a valid address in a grove which has had O_y happen to it.

Each of these corrections requires the same process - one chooses a subject address to be corrected, and the address of a perturbation which applies to the grove, does a vector comparison between them, and applies an appropriate correction to the subject address. For concreteness, consider the first correction to $s_x{}'$ above, and let $O_x = \texttt{delete}(\{\texttt{A}, (0, 1, 1)\})$ and $O_y = \texttt{delete}(\{\texttt{A}, (0, 0)\})$. The effect of operation O_y on addressing is illustrated in figure 4. A correction to any subject address ($s_x{}'$ in this case) is necessary iff:

- The subject address and the perturbation address (s_y in this case) have a common tree identifier - they point to the same tree.
- The paths of the subject and perturbation addresses have a common prefix - they point to locations in the same branch of the same tree.
- The suffix of the path of the perturbation address, after the common prefix, is at most one index.

The latter two conditions are detected by making a vector comparison between the paths $s_x{}'$ and s_y. Starting from the index nearest the root, corresponding pairs of indices are compared. Comparison stops when two indices are found to differ. The first pair of indices is 0 and 0. These are the same, so both paths point to the same branch. The second pair of indices is 0 and 1, which are different, and s_y

finishes there. Inspection of figure 4 shows that ${s_x}'$ should be $\{\mathtt{A}, (\mathtt{0}, \mathtt{0}, \mathtt{1})\}$. That is, the component of the path vector next to where s_y terminates is adjusted by subtracting one.

Some other possible outcomes of a vector comparison are illustrated in figure 6. If the subject address points to $N3$, and the perturbation address points to $N6$, the perturbation to the grove will have no effect on $N3$. A perturbation at $N2$ would likewise have no effect on $N5$. Conversely, if $N5$ is a source for a `move`, then the entire branch containing $N2$ has moved and this has to be considered when rewriting the address for $N2$ for operations after that one. There are in all nine cases to be considered for each comparison.

The derivation of ${d_x}'$ is similar to that for ${s_x}'$. Again, we set the new address equal to the original address (${d_x}' = d_x$). Again, we make a series of vector comparisons to determine what corrections to ${d_x}'$ are required. In order, we compare ${d_x}'$ against

- s_x to exclude the effect of removing N from the grove,
- s_y to include the effect of removing M from the grove,
- d_y to include the effect of replacing M elsewhere in the grove, and
- ${s_x}'$ to re-include the effect of removing N from the grove, at the source address as adjusted for the impact of O_y.

${d_x}'$ is now a valid destination address for ${O_x}'$ which points to a location in a grove which has had all of O_y happen to it. With the new source and destination addresses, plus the details copied from the old O_x, the new ${O_x}'$ can be constructed.

CONCLUSIONS AND FUTURE WORK

This paper has proposed that the REDUCE collaborative editing framework be adapted to the grove data model, in order to create a general solution to the synchronous collaborative editing problem. We have discussed some considerations in the construction of such a solution. Our immediate priorities for future work are

1. Formal proof of correctness and complexity analysis for the algorithms,
2. Resolving outstanding implementation issues and building a testbed prototype, and
3. Conducting field trials of the prototype.

We plan to build a prototype for this technique by modifying the open source editor Amaya, which is flexible and rich in functionality. Amaya's data model is a forest of trees, and Amaya supports editing of XHTML (hypertext), SVG (vector graphics) and MathML (mathematics). This will enable us to test the generality of our solution. The field trials would assess the usefulness of the prototype as a tool for collaboration over documents, and the specific advantages of a general solution to the collaborative editing problem over a suite of solutions targetting the separate kinds of content listed above.

1. REFERENCES

[1] 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.

[2] 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.

[3] ISO/IEC 8879:1986 Standard Generalized Markup Language (SGML), 1986.

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

[5] Peter Murray-Rust, Henry S. Rzepa, and Michael Wright. Development of Chemical Markup Language (CML) as a System for Handling Complex Chemical Content. *New Journal of Chemistry*, 25(4):618–634, March 2001.

[6] R. Karp and V. Chaudhri and J. Thomere. XOL: An XML-Based Ontology Exchange Language, July 1999.

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

[8] 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.

[9] 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.

[10] XML Topic Maps (XTM) 1.0 Specification. World Wide Web, http://www.topicmaps.org/xtm/1.0/, 2001.

[11] Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation, http://www.w3.org/TR/REC-rdf-syntax/, 1999.

[12] XHTML 1.0: Extensible Hypertext Markup Language. World Wide Web, http://www.w3.org/TR/xhtml1/, 2000.

[13] Mathematical Markup Language (MathML) 2.0. World Wide Web, http://www.w3.org/TR/MathML2/, 2001.

[14] Scalable Vector Graphics 1.0 Specification. World Wide Web, http://www.w3.org/TR/SVG/, 2001.

[15] Synchronized Multimedia Integration Language (SMIL 2.0). W3C Recommendation, http://www.w3.org/TR/smil20/, 2001.

```
<!DOCTYPE simple-xhtml SYSTEM "simple.dtd">
<HTML>
<OL id="Conclusions">
 <LI priority="5">The Net should have names for things.</LI>
 <LI priority="10">Namespace collisions <EM>should</EM> be avoided.</LI>
 <LI priority="4">URLs avoid namespace collisions.</LI>
</OL>
<OL id="Recommendations">
 <LI priority="3">We should name things using URLs.</LI>
 <LI priority="2">We should find things using URLs.</LI>
</OL>
</HTML>
```

Figure 1: An example XHTML document, `example.html`

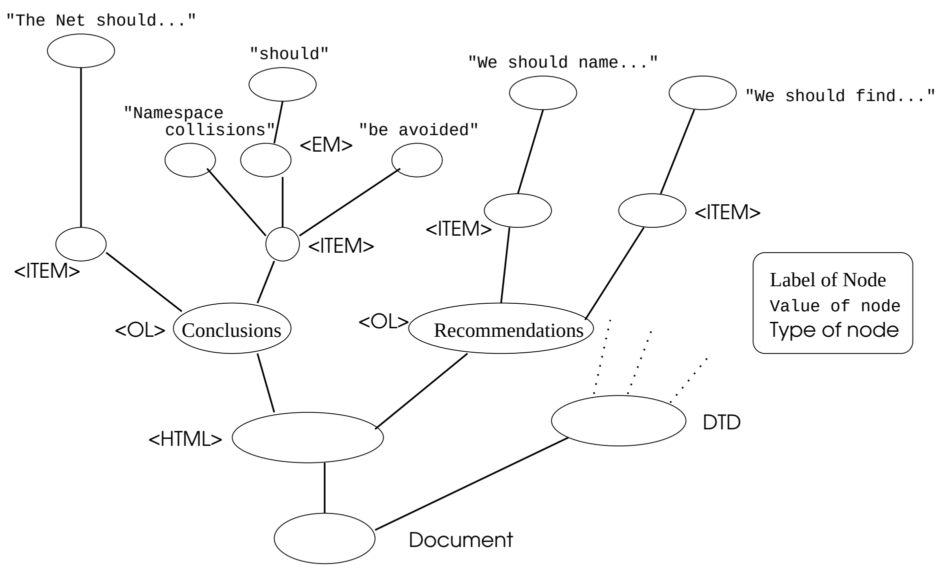

Figure 2: A complete primary grove for `example.html`

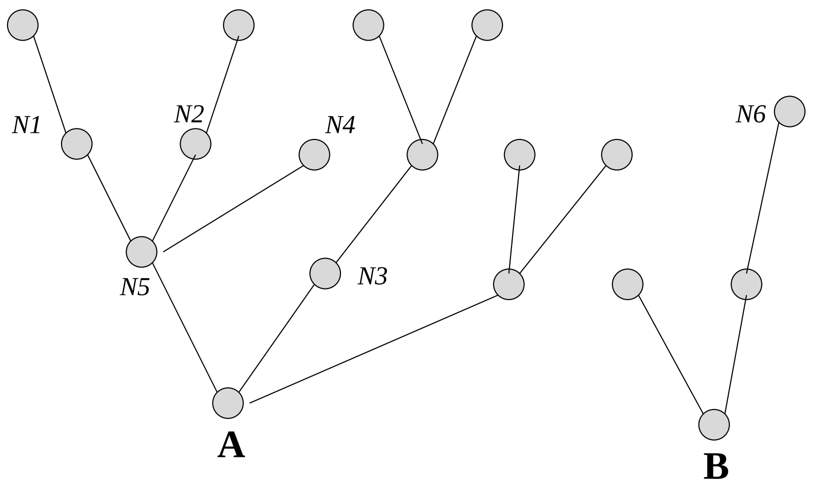

Figure 6: Possible relationships between grove addresses