Three Perspectives on Change

Daniel Wagner

January 21, 2011

Abstract

We investigate the problem of formalizing and modeling edits to tree-based data structures, evaluating the relative merits of three different approaches inspired by version control systems and database management. The first provides a theoretical backdrop, anchoring the discussion; the second boasts simplicity as well as a tractable algorithm for computing an approximately minimal edit; the third allows for great expressiveness, but sacrifices the tractability of some problems.

1 Introduction

Early programs were short and static. The infrastructure made frequent changes impractical (at the earliest, not only were punch cards hand assembled, but the bits themselves were hand written), and hardware limitations forced programs to be minimal both in code size and complexity. But progress was speedy; the burgeoning size of hard drives, processor memory, and processor speed enabled first assemblers that allowed more frequent code changes, then compilers that allowed larger code bases, then IDEs that integrated compiler tools into the text editor and made frequent, sweeping changes to the code-base possible and easy.

Sharing large, highly-variable code bases soon became quite a chore; keeping everybody synchronized became a serious logistical problem. Sending everybody involved an entire copy of an updated code base was impractical. Various heuristics for computing *diffs* were developed, culminating in the well-known Hunt-McIlroy algorithm for computing longest common subsequences. [13] This algorithm was then used in the development of *revision control* tools that eased the sharing and tracking of these diffs.

This was one of the first times edits were considered as first-class entities in computer science, but by no means the last. Below, we will focus on three major modern users of first-class edits: revision control systems, text editors, and databases.

As mentioned above, revision control systems have a history that is deeply intertwined with that of diff-like tools for computing longest common subsequences. Early systems like RCS, CVS, and (to a somewhat lesser extent) Subversion use diff to track changes to each file in a repository separately. However, over time, the consensus among revision control system designers has begun to

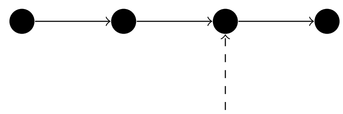

Figure 1: A simple linear edit history

lean towards the idea that the system should track changes to an entire codebase as a whole; that is, that changes to a single file are meaningless outside the context of the remainder of the codebase.

For example, one common task in a C-based project would involve adding a function declaration to a header file `foo.h` and a corresponding function definition to an implementation file `foo.c`. Using only diff-style edits, these two files are separate objects, tracked separately; the consequence of this is that it is perfectly reasonable in that model to ask the system to roll back the additions to `foo.h` without rolling back `foo.c` in the corresponding way. Expressing the constraint that the two files should be modified together *steps out* of the diff model of tracking simple sequences.

To mimic a filesystem, a useful model for these systems treat *trees* as the data type of interest, and build a model of edits to these trees. The kinds of edits we consider below we will all be of this form; while evaluating them, we will want to keep in mind some of the common tasks that revision control systems are asked to perform:

- Rolling back ill-advised changes
- Detecting changes—that is, given two different trees, finding a smallish edit that changes one to the other
- Reconciling changes made in parallel by different authors
- Tracking the provenance of particular parts of the tree

Tracking edits explicitly can also be useful, on a smaller scale, in text editors. A key feature of editors is its *undo* and *redo* functionality; different editors offer wildly varying levels of support for this. The baseline functionality involves modeling the code as a simple sequence of characters and tracking a “linear” history (as in Figure 1). The undo and redo actions simply move the pointer forward and backward in the history; making a change other than a redo might (for example) discard the future part of the history.

There are at least two orthogonal directions this baseline can be improved. First, the history itself can be given additional structure: rather than discarding the “redo future” whenever making a change, the model could simply branch, resulting in a richer edit history than a straight line – perhaps a tree or even a DAG. (Figure 2 gives an example of this.) It then becomes natural to ask whether changes from one branch can be migrated to another branch. Many

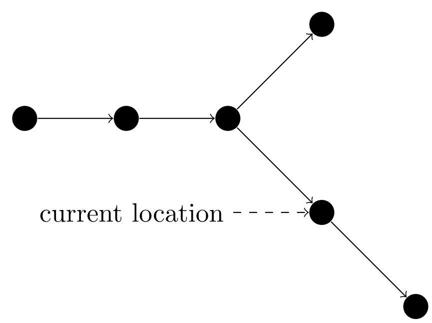

Figure 2: A more complex edit history

papers examine this question [1, 2, 4, 10, 14]; one of the major lessons is that representing edits with rich data about the *intention* of the user makes the algorithms much more predictable and usable.

Thus, a second possible axis of improvement, which we will focus on in this paper, is modifying the edits to operate on parse trees rather than simple character sequences. Note that the requirements for these tree edits are somewhat different from the requirements cited for revision control systems:

- Rolling back ill-advised changes
- Visualizing changes in a human-readable way
- Relocating changes within the history meaningfully

In particular, computing edits given two trees is no longer a key feature; the editor itself can track the *actual* edits made by the user. Tracking provenance is likewise unneeded.

The final application area we will consider is that of an XML-backed database, backing, say, a web application. The edits, therefore, are in fact a form of communication between the web application and the database, rather than being manipulated in memory (or on disk) from within a single program, as in revision control systems or text editors. Moreover, the bulk of the edit will be hand-composed by a human (namely, the programmer developing the web application) – though it may have some holes or variables to be filled in at run time. For our purposes, we may well assume that an edit will be executed much, much more frequently than it is composed.

As a result, the requirements of an edit language for databases are radically different than those for revision control systems or text editors. In particular, a web application often has multiple users, and consequently may submit many edits to the database simultaneously; the database should behave as though they were submitted in some order (though it may execute them concurrently). Combining these characteristics, we find that doing some one-time analysis of edits can be very worthwhile if the results of the analysis can be used to speed up the application of those edits at runtime, because the benefits of the analysis will be reaped many times, but the cost paid only once. So a successful edit language should have these properties:

- Human-writable, text-based syntax
- Compact representation for inter-program communication
- Speedy concurrent apply operation (perhaps via a helpful static analysis)

(Note that in particular this application may not place such high priority on the ability to roll back changes.)

There is unending variety in edit languages. Indeed, even when considering the data objects that the edits are intended to modify, there are a huge number of different models possible: binary (opaque) data, sequences, sets, trees, relations, DAGs, graphs, and more. Each of these could presumably have a whole collection of reasonable choices of edits from simple to expressive. On one end of the spectrum, we have the diff format for editing sequences; on the other end lies programs written in (say) C that edit any data type we could imagine; somewhere in the middle lie relations and the relational calculus. Surveying all the known edit languages is well beyond the scope of this paper. Instead, we will consider only node-labeled, unordered trees as our data model, and consider just a few languages that cover a range of the expressiveness spectrum.

We will also narrow our focus to only three edit languages. The Löh, et al paper strives to introduce a framework for reasoning about the behavior of revision control systems [15]. It will include more precise definitions for words we have so far been using informally: edits, edit history, conflict and compatibility, merging, synchronization, and so forth. The second language (as described in the Chawathe and Garcia-Molina paper [6]) is designed to include an expressive enough core that machine-generated edits can encapsulate the "idea" or "meaning" of a change while keeping tractable computation of a (nearly) minimal diff between old and new trees. Finally, the third language (introduced in the Ghelli, et al paper [12]) is designed to make deciding whether two edits depend on each other—to be more precise, whether they commute—tractable, while emphasizing the inclusion of high-level features designed to make the language human-writable.

Before diving in, we briefly describe a few typographic and metasyntactic conventions. Variables d and e will correspond to concrete pieces of data and edits, respectively. Edits will have a partial function **apply** associated with them; $\mathbf{apply}(e, d) = d'$ indicates that edit e transforms data d to data d', and $\mathbf{apply}(e, d) = \perp$ indicates that edit e is incompatible with data d. The definition of **apply** will vary between sections. Additionally, we will assume an infinite supply L of labels; we will use metavariable ℓ for an element of this set. We will use $\triangle$ for the symmetric difference and $\#$ for set disjointness.

2 Semantics and conflict detection

We will begin our discussion by developing a formal framework in which we can define precisely terms like *edit history*, *edit*, *working copy*, *merge*, and *conflict*. (We will also sometimes use *repository* interchangeably with edit history, and *patch* interchangeably with edit.) The aim of the current exercise is to develop a semantic model we can use to unify a variety of edit languages, following the development of Löh, et al [15].

2.1 Data and edits

As mentioned above, we will use node-labeled trees as our data model; in fact, we will demand that node labels have two parts: a *name* drawn from L and some *contents* drawn from another arbitrary set C. (For example, when this is modeling a file system, L would be the set of valid filenames, and C would include information like permissions and the bytes in the file.) Moreover, we will use a somewhat idiosyncratic representation of these trees; we will defend the idiosyncrasies in Section 2.3.

For our purposes, a tree is a set of assertions A, where each $a \in A$ has the form

$$\begin{aligned} a &::= \ell_p \to \ell_c \mid \ell \textbf{ contains } c \\ c &::= \text{(elements of C)}, \end{aligned}$$

and where A conforms to the well-formedness constraints given in Figure 3. We assume the existence of a distinguished label **root** $\in L$, and adopt the convention that variables free in the premise of an inference rule are bound universally, while variables free in the conclusion (that are not bound in the premise) are bound existentially. Rule Parent says that each node has at most one parent. The UniqC rule tells us that each node is labeled with at most one piece of content. Rules Full1, Full2, and Reachable together say that every node is labeled and reachable from **root**. Finally, rule Cycle rules out cycles.

At its core, an edit to this structure is simply an object (S, T) with a set of assertions S to remove and a set of assertions T to add. We could then say that the patch is applicable to d when $S \subset d$, that is, when all of the assertions that the patch aims to remove are available.

However, this proposal has a small flaw: it makes no mention of the invariants of d. Indeed, there are certain kinds of patches which, no matter how we construct them, cannot guarantee that it maintains the invariants whenever it is applicable. Consider, for example, the edit that simply inserts a new top-level node at ℓ with contents c, namely $(\emptyset, \{\ell \textbf{ contains } c, \textbf{root} \to \ell\})$. In most trees, this is fine, but if the top-level node ℓ already exists with contents other than c, then applying this patch would break UniqC. Thus, we would like a way to say that certain assertions do not exist in the data, namely, any assertion assigning contents to ℓ.

$$\frac{\begin{array}{c}\ell_p \to \ell_c \\ \ell'_p \to \ell_c\end{array}}{\ell_p = \ell'_p} \quad \text{(Parent)}$$

$$\frac{\begin{array}{c}\ell \textbf{ contains } c \\ \ell \textbf{ contains } c'\end{array}}{c = c'} \quad \text{(UniqC)}$$

$$\frac{}{\textbf{root contains } c} \quad \text{(Full1)}$$

$$\frac{\ell \to \ell'}{\ell' \textbf{ contains } c} \quad \text{(Full2)}$$

$$\frac{\ell \textbf{ contains } c}{\textbf{root} = \ell_0 \wedge (\forall i \in \{0, \ldots, n\}.\ell_i \to \ell_{i+1}) \wedge \ell_n = \ell} \quad \text{(Reachable)}$$

$$\frac{\forall i \in \{0, \ldots, n\}.\ell_i \to \ell_{i+1}}{\ell_0 \neq \ell_n} \quad \text{(Cycle)}$$

Figure 3: Invariants that guarantee tree-structure

We can solve this problem by adding a third element to the definition of a patch: a set of forbidden assertions. We thus arrive at our final definitions for patch and applicability:

Definition 1. *A* patch *or* edit *is a triple* (S, E, T) *where* $S \cup T \subset E$*.*

We will sometimes write $(S, \_, T)$ when we mean $(S, S \cup T, T)$.

Definition 2. *An edit* (S, E, T) *is* applicable *to data* d *when* $E \cap d = S$*, and*

$$\textbf{apply}((S, E, T), d) = \begin{cases} d \triangle S \triangle T & (S, E, T) \textit{ is applicable to } d \\ \perp & \textit{otherwise} \end{cases} .$$

(It is worth pointing out that $E \cap d = S$ implies, in particular, that $S \subset d$.)

Revisiting the example above, we can now write

$$e_1 = (\emptyset, \{\textbf{root} \to \ell\} \cup \{\ell \textbf{ contains } c' | c' \in C\}, \{\textbf{root} \to \ell, \ell \textbf{ contains } c\})$$

for the patch that adds a new top-level node. Unlike the previous proposal, we can now state with confidence that whenever e_1 is applicable to d, the application $\textbf{apply}(e_1, d)$ will preserve the invariants of our tree structure. Of course, we can still write patches that will violate the invariants; however, what we can do with this definition of a patch that we could not do with the first proposal is we can write patches for which we can prove that they will never violate the invariants when they are applicable.

For convenience, we will adopt the following abbreviation. When the iteration domain is clear from context, we will write $*$ as a placeholder for all values in that domain; so $\{\ell \textbf{ contains } c' | c' \in C\}$ could be shortened to $\{\ell \textbf{ contains } *\}$, for example. Each appearance of $*$ is independent. Using this notation, the above patch would look like this:

$$e_1 = (\emptyset, \{\textbf{root} \to \ell, \ell \textbf{ contains } *\}, \{\textbf{root} \to \ell, \ell \textbf{ contains } c\})$$

As another example patch, consider adding a node lower in the tree, as a child ℓ_c with contents c_c of ℓ_p (which has contents c_p). To preserve the invariants, we must make sure not only that there is no node already at ℓ_c, but also that the parent node ℓ_p exists. We can achieve this by putting ℓ_p **contains** c_p in both S and T, that is, by requiring ℓ_p **contains** c_p to be in the data before application, and to include it after application.

$$\begin{aligned}
S_2 &= \{\ell_p \textbf{ contains } c_p\} \\
E_2 &= \{\ell_p \textbf{ contains } c_p, * \to \ell_c, \ell_c \textbf{ contains } *\} \\
T_2 &= \{\ell_p \textbf{ contains } c_p, \ell_p \to \ell_c, \ell_c \textbf{ contains } c\} \\
e_2 &= (S_2, E_2, T_2)
\end{aligned}$$

2.2 Constructors for common patches

With this framework in place, we can define some syntax for common tree operations. None of the theory in the following parts depend directly on the existence of this syntax or their interpretations, but they serve as a good way of illustrating the concepts introduced so far. We will define syntax for four operations: node insertion, updating the contents of a node, reparenting or moving a subtree, and copying a subtree.

For insertion, we will allow the inserted node to "steal" some of its siblings to use as children. That is, we will specify a label and contents for the new node, a parent for the new node, and a collection of the parent node's children that should become children of the new node. We will also restrict the nodes that we insert to ones not already in the tree. Thus:

$$\begin{aligned}
\textbf{insert}(\ell_p, \ell_c, c_c, L_c) = (&\{\ell_p \to \ell \mid \ell \in L_c\}, \\
&\{\ell_p \to \ell \mid \ell \in L_c\} \cup \{\ell_c \to \ell \mid \ell \in L_c\} \cup \{* \to \ell_c, \ell_c \textbf{ contains } c_c\}, \\
&\{\ell_p \to \ell_c, \ell_c \textbf{ contains } c_c\} \cup \{\ell_c \to \ell \mid \ell \in L_c\})
\end{aligned}$$

Updating the contents of a node and moving subtrees are perhaps the easiest of the patches to represent in our framework:

$$\begin{aligned}
\textbf{update}(\ell, c_{old}, c_{new}) &= (\{\ell \textbf{ contains } c_{old}\}, _, \{\ell \textbf{ contains } c_{new}\}) \\
\textbf{move}(\ell_c, \ell_p, \ell'_p) &= (\{\ell_p \to \ell_c\}, _, \{\ell'_p \to \ell_c\})
\end{aligned}$$

Copying a subtree is a bit more involved. We must first identify a subtree; let us say that S is a set of assertions that satisfy the invariants given above (but with a different distinguished **root**; call it $\mathbf{root}_S$). To copy this subtree, we must choose new node labels. Suppose $f : L \to L$ is a relabeling function. Lift f to an assertion relabeling function:

$$f(\ell\ \mathbf{contains}\ c) = f(\ell)\ \mathbf{contains}\ c$$
$$f(\ell \to \ell') = f(\ell) \to f(\ell')$$

Lift f to sets of labels and sets of assertions in the natural way. We will also say that ℓ in S when $\ell \to \ell' \in S$ or $\ell' \to \ell \in S$ or ℓ **contains** $c \in S$. Then we can define

$$\begin{aligned}\mathbf{copy}(S, \ell, f) = (&S,\\ &S \cup f(S) \cup \{* \to \ell_c | \ell_c \text{ in } f(S)\},\\ &f(S) \cup \{\ell \to f(\mathbf{root}_S)\})\end{aligned}$$

for injective f. The additional constraint $\{* \to \ell_c | \ell_c \text{ in } f(S)\}$ ensures that all of the labels generated by f are actually new for the tree.

As a bonus, patches inverse gives us a node deletion operation for free (by inverting **insert**), as well as a slightly weirder *glue* operation (from inverting **copy**) which merges two identical subtrees, causing one to disappear.

2.3 A comment on flexibility

It is worth taking a moment to address the question that is no doubt on the reader's mind: why are we using such a weird representation of trees?

We could certainly have stated all of the definitions given so far in a more familiar way: we could represent trees in the usual way as a collection of nodes and edges, with a labeling function mapping node identifiers to their contents. The definition of what a patch is would have to be modified, as well, to include more specific notions of nodes and edges, ways of modifying the labeling function, and allow for more tree-specific preconditions. (For example, we might choose analogs to the syntax given above as the primitive edits, or some variant of them.)

The result, however, would have the definition of patch hopelessly interwoven with the particular kind of data that patch is modifying. The presentation here has the much nicer property of being data-agnostic when defining the meaning and applicability of patches. As a result, the definitions above (and the definitions below) can be used verbatim even if the data model changes: the ideas here apply equally well to different data structures.

Of course, some things change. For different data structures, we must choose a different collection of basic assertions, and rework the invariant that the whole collection of assertions must satisfy. However, these extensions are generally quite self-contained.

2.4 Repositories and edit histories

The least well-understood ideas in revision-control system are those of merging and conflicts. To work towards understanding these ideas, we now attempt to define properly what a repository is and how repositories can interact.

Definition 3. *A* working copy *is a tree (represented as described above).*

Definition 4. *A* repository *is a bag of patches.*

These definitions are deceptively simple. For example, the simplest question we might ask is, "How are repositories and working copies related?". We can build up a working copy by applying a sequence of patches to the empty set of assertions. However, given only a repository, we are at something of an impasse: the patches are unordered, and we are not at all guaranteed that if we choose an arbitrary order that application will be well-defined, or even that any ordering exists that makes application well-defined. We will therefore call a repository *consistent* when there is an ordering of the patches $\langle e_0, \ldots, e_n \rangle$ such that, given $d_0 = \emptyset$, each $d_{i+1} = \mathbf{apply}(e_i, d_i)$ is well-defined, that is, $d_{i+1} \neq \bot$. It is convenient that if two such orderings exist, they result in the same final working copy (thanks to the commutativity of $\triangle$).

With that concept in hand, we can define merges and conflicts.

Definition 5. *Given two consistent repositories R_1 and R_2 and a set of patches $P \subset R_2 \setminus R_1$ that we would like to transport from R_2 to R_1, we say that $R_1 \cup P$ is successfully* merged *when $R_1 \cup P$ is consistent, and that it is in* conflict *otherwise.*

2.5 Discussion

We now have a firm definition for many of the ideas we will discuss in this paper. The definitions are concise; they are elegant in their coverage and orthogonality. However, there are a few significant shortcomings of the development so far.

The first is that a direct translation of these ideas to code would be impractical. Many of the common patches discussed above involve manipulating infinite sets, which makes the problem a non-starter. Of course, this can be combated in most cases by judicious choice of the representation of a patch. For example, some algebraic data structure involving the patch constructors above, patch composition, and patch inverse might be reasonable. The role of the work discussed here, then, is to provide a model against which to verify the manipulations of these higher-level representations. Similarly, a real implementation would need a similar process for designing a more concrete representation of repositories and working copies, appealing to the simple set-based model when verifying algorithm correctness.

A second serious drawback is the simplicity of the working copy that we used. However, this drawback is not inherent to the approach. Indeed, the exposition by Löh, et al demonstrate extensions to the model that allow for line-based files, repository meta-data, and tagging. Some similar extensions are likely possible for tracking patch ordering information, if that becomes desirable.

Finally, we have so far seen only very idealized definitions, and no practical algorithms. For example, we have not discussed any practical way of checking even the consistency of a given repository! In practice, we would also want algorithms for computing minimal patch sets that successfully merge and for comparing two working copies. These may be quite difficult in practice.

We can revisit the criterion discussed in Section 1 for revision control systems, text editors, and databases to evaluate how far this approach takes us.

One feature we want in revision control systems is the ability to roll back ill-advised changes. The current approach actually gives us *two* ways of doing this. The first is to add an inverse patch[1] to the repository; this records that we made an edit and later decided it was a bad idea. The second is to remove the patch from the repository entirely; we can do this if we want to pretend that we never did that edit in the first place. So that criteria seems well-covered.

We would also like an algorithm for detecting changes; the current approach does not provide such a thing, but the approach of the next section attempts to handle this. For reconciling changes made in parallel, the current approach seems to give us a good definition of what it would mean to do this (that is, a successful merge reconciles different repositories), but gives no hint of how to implement a tractable merge algorithm. (This does not seem to be a fundamental limitation of the approach – but simply something that has not *yet* been solved.) Finally, the problem of tracking provenance is tackled somewhat in the paper (though the details have been omitted here for space).

For text editors, in addition to rollback, we also want visualizations and change relocation. While a good visualization is likely possible, none of the discussion so far has focused on it; getting this right would likely take some serious work. On the other hand, relocating changes within the history is almost trivial in this framework. Since a repository is an unordered set of patches, relocating a patch is the identity!

So this approach is a mild success for revision control and text editing; but it seems to fail all three criteria for a database system. Patches of this form are not particularly human-writable; as mentioned above, some serious thought would be required even to represent these patches in memory; and finally, there is no clear provision for concurrency. We will need another approach for databases.

3 Change detection

3.1 Problem description

In this section, we will develop one of the missing algorithms, following the development of Chawathe and Garcia-Molina [6]. The problem statement is fairly simple: we have two working copies (say, one from a repository and one

[1] The inverse of (S, E, T) is (T, E, S), of course.

from the disk), and we would like to generate a patch to convert one working copy to the other.

Actually, the problem as stated is quite simple: given working copies S and T, the patch $(S, \_, T)$ satisfies the criteria. It is nevertheless clearly not the desired patch. To narrow the problem, we introduce a cost function for patches; we then wish to find not just any patch, but a minimal-cost patch. We will allow sequences of patches of the simple forms introduced in the previous section[2].

Dozens of variants of this problem have been studied [5, 7, 8, 19]. Points of variation include:

- The exact details of how each edit behaves often differ, though they are usually fundamentally similar to the ones proposed above.
- Different approaches consider cost functions of varying sophistication.
- Some analyses assume that nodes have identities which can be compared between working copies, and others do not make this assumption.

One of the major interactions between the first two variations involves how the cost function behaves on moves and copies. Some analyses simply disallow moves and copies (which is tantamount to simply making them prohibitively expensive compared to deleting and inserting nodes). These analyses benefit from much greater simplicity; however, the edits generated by them can be somewhat surprising to human readers. On the other hand, making moves and copies cheap enough that they must be considered during the analysis elevates the problem to NP-hardness.

For now, we will make the following decisions:

- As mentioned above, we will use exactly the edits described in Section 2.
- We will use a fairly simple cost function: we will have constants c_i, c_d, c_m, c_c, and c_g for the costs of **insert**, inverse **insert** (i.e. deletion), **move**, **copy**, and inverse **copy** (i.e. glue), respectively. For the cost of an **update**, we will allow an arbitrary function $c_u : C \times C \to \mathbb{R}$ to give the cost given old and new contents.
- We will not assume that nodes have persistent identities; the detection algorithm will have to discover correspondences on its own.

In particular, the costs of **move** and **copy** will likely be quite low compared to moving and copying entire subtrees. To avoid exponential running time, the algorithm presented will have to make approximations in a few places; we will point them out as they arise.

The solution we present has two main pieces: first, finding an *alignment* or correspondence between the nodes of the two working copies, and then converting that alignment into a low-cost edit.

[2]We can **apply** a patch sequence by using **apply** on the composition of the patches in the sequence. Deriving the triple for the composition of two patches is easy, but tangential.

3.2 Alignment discovery

Define the operator $(-)^+$ to add a fresh, newly-labeled, disconnected node to a tree. (The contents of the node are immaterial for our purposes.) Below, we will assume the freshly chosen label is $+$.

Definition 6. *An* alignment *between d_1 and d_2 is a minimal edge cover of the complete (unweighted) bipartite graph whose parts are the node labels of d_1^+ and the node labels of d_2^+.*

Recall that an edge cover of a graph is a subgraph with all the nodes and in which each node has degree at least one. A minimal edge cover is one which achieves this with the minimal number of edges (for unweighted graphs) or at minimal cost (for weighted graphs). Thus, an alignment gives a correspondence between the nodes of the two graphs via its edges: if there is an edge (a, b) in the alignment, we read this as meaning that the edit we compose should turn node a in d_1 into node b in d_2. Similarly, an edge $(a, +)$ indicates that the edit we compose should delete a from d_1, and an edge $(+, b)$ indicates that the edit should create b from scratch.

We will assign to each alignment the cost of the patch that results from that alignment, and our goal will be to find a minimal alignment. We must therefore turn to the problem of converting an alignment into a patch. What we will hope to do is to assign a weight to each edge of the complete bipartite graph in such a way that choosing a minimal cover of the *weighted* graph corresponds to choosing a minimal alignment.

3.3 Converting from alignment to edit

One algorithm involves simply converting each edge in the alignment to a few atomic updates, as well as some global ordering constraints on which edges' updates should come earliest. It proceeds by case analysis on the edge:

- $(+, +)$: No edit is needed for this edge.
- $(a, +)$: Check which other nodes have edges to $+$. If $c_g < c_d$, that is, inverse **copy** is cheaper than inverse **insert**, then look for another node with the same label which we can glue to. If so, generate an inverse **copy**, possibly prefixed by an **update**, and a constraint saying that the deletion of the other node must come after these edits. If no suitable glue operation exists, generate a deletion.
- $(+, b)$: Similarly to the previous case, check which other nodes have edges from $+$. If $c_c < c_i$, that is, **copy** is cheaper than **insert**, then look for another node with the same label that we can copy. If so, generate a **copy**, possibly followed by an **update**, and a constraint saying that these must come after the creation of the node we are copying. If no suitable copy operation exists, generate an **insert**.

- (a, b): By far the most complicated of the bunch. Naively, this would be a **move** or a **copy** possibly followed by an **update**. However, we should first walk up the tree, so that we **move** or **copy** the highest possible node; that node's subtrees will then get moved or copied "for free".

  In fact, there is an additional wrinkle. Suppose there is a **copy** operation high in the tree, and another operation **move**s a node somewhere in the subtree that is being copied. We may now choose whether the copy that we create includes or does not include that node by choosing to perform the **move** before or after the **copy**. Thus, the algorithm must also include some logic which evaluates such situations (called "free copies" in the Chawathe, et al exposition) to minimize the number of **move** and **copy** operations; we will skip the details of this logic.

After this algorithm runs, the edges are topographically sorted (using the constraints generated as the topography), and the atomic edits are sequenced in the resulting order.

The algorithm itself may sound somewhat complicated, but the take-away message is fairly simple: there is no such thing as the "cost of an edge" in the complete bipartite graph. The cost of the updates generated by any particular edge in this graph depends on what other edges exist in the particular alignment chosen. As a result, we cannot simply give weights to the edges of the complete bipartite graph as we had hoped.

The development by Chawathe and Garcia-Molina adopts a slightly more subtle approach: each edge is given an upper and lower bound on the cost that could be incurred by including a particular edge. It is then possible to prune away obviously bad edges; as the complete graph is reduced, the bounds are improved, making more pruning possible. We will skip the algorithms involved in computing the initial bounds, pruning bad edges, and updating the bounds as the graph evolves; they are clever, but not central to the idea.

When this process terminates, we have a small, bipartite graph, where each edge is labeled with upper and lower bounds. We can use the lower bounds as estimates of the cost each edge contributes to the alignment and run a standard weighted matching algorithm to find an approximately minimal cover. Given this cover, we can then generate the edit corresponding to the cover, as described above.

3.4 Discussion

Before addressing the strengths and weaknesses of this approach, we pause to explicitly identify the places in the algorithm where approximations are made (leading to occasionally sub-optimal edits). First, the final edit we create is generated from an alignment, and our algorithm for converting alignments into edits is not surjective. For example, any single node in the tree gets assigned at most one of the **insert**, inverse **insert**, **move**, **copy**, or inverse **copy** edits and at most one **update** edit. Furthermore, the generated edit will never modify nodes that are in neither tree – for example, by creating a new node, doing something with it, and then deleting it. Finally, no generated edit will perform an inverse **copy** on any node involved in a normal **copy**, either directly or indirectly. Each of these restrictions may seem spurious, but in fact violating them can be useful; thus, failing to generate edits that violate them is approximation number one. Still, whether this approximation is good or bad is somewhat philosophical; the edits that take advantage of the extra abilities outlined above are often somewhat tricky, and may be surprising to a human reader.

The second approximation comes at the end, when choosing a minimal cover in the weighted bipartite graph. Since there is no true weight for the edges in the graphs, the labeling chosen (for example, using the lower bounds) must be an approximation. In some cases, there may be few enough possible covers that we can perform an exhaustive search, but often this will not be the case. Luckily, this is an approximation with a knob that we can twiddle: we can improve our edits (at the cost of runtime) by examining more possible covers and choosing the minimal one out of the ones we examine.

The algorithm outlined above is admittedly quite complex. Because of the two approximations mentioned, it is also somewhat heuristic – there are no theoretical guarantees about its output, and so it could potentially have somewhat unpredictable behavior. Additionally, the complexity makes this approach somewhat inflexible to variations in the set of edits. The conversion between alignments and minimal edits as well as the cost estimation algorithms are fairly dependent on using exactly the set of edits described above.

Despite these shortcomings, this approach also has several nice properties. It is fairly fast; for n nodes, it is $O(n^3)$ worst-case, and experimentation suggests that it is $O(n^2)$ for the typical case. The cost model proposed is fairly flexible, as well. Moreover, detecting node moves and copies is a nice feature. For example, in the revision control system domain, detecting moved or copied files is key for maintaining provenance information. Many current systems either require the user to manually specify moves and copies or apply some easily-fooled ad-hoc heuristics for detecting them. Reliably analyzing this kind of code motion could be very beneficial for these systems.

Thus, this approach remedies one of the shortcomings of the previous approach for use in a revision control system. Additionally, it makes some small strides in the text-editor and database domain by providing a compact, concrete representation for edits. It is possible that either the alignments discussed above or the actual edit sequences computed could be readily visualized by a human; similarly, it seems much more feasible for a human to construct one of these edits and for programs to exchange these edits than than it was for edits of the form discussed in Section 2. Still, it leaves much to be desired for human-writability; we will need to consider another approach for our edits to be sufficiently high-level.

4 Reordering

In the previous section, we discussed an edit language designed primarily for machine generation, with a passing interest in human readability. As a result, we ended up with a language that had a few very low-level editing capabilities, and the only way of creating large-scale edits was by chaining together many, many small-scale edits. In this section, we will instead focus on a language designed for human generation. It will feature high-level abstraction mechanisms for iteration and scoping, allowing for concise descriptions of large-scale edits. As a trade-off, the kind of analysis described for finding minimal edits achieving a certain effect will likely be impossible or intractable; nevertheless, another kind of analysis for detecting whether two edits are sensitive to application order is still possible.

This language was designed from the perspective of database maintenance. There are a great variety of such languages, including myriad variants of SQL (based on the relational calculus [9]), Datalog (for querying deductive databases [17]), XQuery [3] (for querying XML databases) and variants that include updates [18], LINQ [16] (which strives in part to unify several database implementations), and so on. Our focus on tree-based data makes XML the most natural representation choice out of these; correspondingly, the language we describe below will be inspired by XQuery.

Properly describing the behavior of the high-level edit language described in the Ghelli, et al paper [12] involves first defining a low-level edit language much like the one described in the previous section, so we will begin there – in familiar territory.

4.1 Edit history

The data model we consider here is slightly more intricate than before. We assume some primitive form of locations *loc* that can express at least URIs and positions in code, a set of names Q, and a set of possible text content elements T. Assertions look like this:

$$\begin{aligned} a &::= \exists\ell \mid \ell \to \ell' \mid \ell \textbf{ is from } loc \mid \\ &\quad\ \ell \textbf{ has } K \mid \ell \textbf{ named } Q \mid \ell \textbf{ contains } T \\ K &::= \textbf{text} \mid \textbf{element} \end{aligned}$$

The invariants are listed in Figures 4 and 5. The invariants in Figure 4 essentially enforce tree-structure (though it allows for subtrees that are not actually connected to anything, which we will use later during deletion). We can view the set of assertions of the form $\exists\ell$ as being a set of nodes. Then rules DomLoc, DomName, and DomKind say that **is from** assertions, **named** assertions, and **has** assertions only apply to nodes that we know about. Together with rules FuncLoc, FuncName, and FuncKind, this implies that the assertions are actually partial functions whose domain is the set of nodes. Rule TotKind further tells us that the **has** assertions actually represent a total

$$\frac{\ell \textbf{ is from } loc}{\exists \ell} \qquad (\textsc{DomLoc})$$

$$\frac{\ell \textbf{ named } q}{\exists \ell} \qquad (\textsc{DomName})$$

$$\frac{\ell \textbf{ has } k}{\exists \ell} \qquad (\textsc{DomKind})$$

$$\frac{\ell \textbf{ is from } loc \qquad \ell \textbf{ is from } loc'}{loc = loc'} \qquad (\textsc{FuncLoc})$$

$$\frac{\ell \textbf{ named } q \qquad \ell \textbf{ named } q'}{q = q'} \qquad (\textsc{FuncName})$$

$$\frac{\ell \textbf{ has } k \qquad \ell \textbf{ has } k'}{k = k'} \qquad (\textsc{FuncKind})$$

$$\frac{\exists \ell}{\ell \textbf{ has } k} \qquad (\textsc{TotKind})$$

$$\frac{\ell_p \to \ell_c \qquad \ell'_p \to \ell_c}{\ell_p = \ell'_p} \qquad (\textsc{Parent})$$

$$\frac{\forall i \in \{0, \ldots, n\}. \ell_i \to \ell_{i+1}}{\ell_0 \neq \ell_n} \qquad (\textsc{Cycle})$$

Figure 4: Routine invariants for XML-like trees

function. Rules Parent and Cycle are identical to the previous sections, and say simply that each node has at most one parent and that there are no cycles.

The invariants in Figure 5 are slightly more interesting. Rule LocRoot says that only root nodes have source location. Rules ElemNamed1 and ElemNamed2 together say that nodes have a name iff they are element nodes; similarly, TextFull1 and TextFull2 together say that nodes have textual content iff they are text nodes. Finally, rule TextLeaf only allows leaf nodes to be text nodes (that is, text nodes have no children).

From here, we will define a few atomic updates in the edit-triple form described in Section 2. (When there may be some confusion, we will refer to these as atomic edits or low-level edits. We will refer to expressions in the later edit language as high-level edits.) When A is a well-formed set of assertions that does not contain any assertions of the form $\ell \to \ell'$ or ℓ **is from** *loc* – that is, when A contains a set of new nodes to add, possibly with text content or names,

$$\frac{\ell \textbf{ is from } loc}{\forall \ell'.\neg(\ell' \to \ell)} \qquad (\textsc{LocRoot})$$

$$\frac{\ell \textbf{ has element}}{\ell \textbf{ named } q} \qquad (\textsc{ElemNamed1})$$

$$\frac{\ell \textbf{ named } q}{\ell \textbf{ has element}} \qquad (\textsc{ElemNamed2})$$

$$\frac{\ell \textbf{ has text}}{\ell \textbf{ contains } t} \qquad (\textsc{TextFull1})$$

$$\frac{\ell \textbf{ contains } t}{\ell \textbf{ has text}} \qquad (\textsc{TextFull2})$$

$$\frac{\ell \to \ell'}{\ell \textbf{ has element}} \qquad (\textsc{TextLeaf})$$

Figure 5: Interesting invariants of XML-like trees

but no edges or locations – we will allow them to be created:

$$\textbf{insert}_n(A) = (\emptyset, \_, A)$$

Given a node and a location, we can extend the location mapping. The middle precondition ensures that we only give locations to root nodes.

$$\begin{aligned}\textbf{insert}_r(\ell, loc) = (&\{\exists \ell\},\\ &\{\exists \ell, * \to \ell, \ell \textbf{ is from } *\},\\ &\{\exists \ell, \ell \textbf{ is from } loc\})\end{aligned}$$

Given a set of assertions E of the form $\ell_p \to \ell_c$, we can add those edges to the working copy. Below, we will use the set $N = \{\exists \ell_p, \exists \ell_c | \ell_p \to \ell_c \in E\}$ of nodes involved. We will also arrange for the addition of E to avoid creating cycles (though we do not represent this fact in the definition of the edit).

$$\begin{aligned}\textbf{insert}_e(E) = (&N,\\ &N \cup \{* \to \ell_c, \ell_c \textbf{ named } * | \ell_p \to \ell_c \in E\},\\ &N \cup E)\end{aligned}$$

Finally, we allow the deletion of edges. Nodes may be deleted, but only by disconnecting their subtree from the top-level tree of interest – that is, by deleting the edge from their parent – never by removing them from the working copy. So, letting E be a set of assertions of the form $\ell_p \to \ell_c$ and using the N abbreviation as before, we have

$$\textbf{delete}_e(E) = (E \cup N, \_, N)$$

When we are writing an edit for a particular piece of data d, we will adopt the notation $\mathbf{delete}_e(S)$ to extract all the edges pointing to nodes in S from the data and delete them, that is,

$$\mathbf{delete}_e(S) = \mathbf{delete}_e(\{\ell_p \to \ell_c \in d \mid \exists \ell_c \in S\})$$

Later, the programs we write in the high-level language will be run on a particular working copy to produce a sequence of these low-level edits, and it will be this pairing of a working copy and edit sequence that we are interested in analyzing.

Definition 7. *A* store history η *is a pair* $(d, \bar{e})$*, where* $\bar{e}$ *is a sequence of edits* $\langle e_1, \ldots, e_n \rangle$ *to be* **apply***'d in turn.*

We will occasionally abuse notation, and write $f(\eta)$ when we really mean $f(d, \bar{e})$ – especially when $f = \mathbf{apply}$.

4.2 High-level edit language

Without further ado, the edit language we will be analyzing:

$$\begin{aligned}
e &::= x \mid e/A :: T \mid e, e \mid \mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{return}\ e \mid \\
&\quad\ \ e = e \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid \mathbf{element}\ \mathit{loc}\ \{e\}\ \{e\} \mid \\
&\quad\ \ \mathbf{delete}\ e \mid \mathbf{insert}\ e\ \mathbf{into}\ e \mid \mathbf{let}\ x := e\ \mathbf{in}\ e \\
A &::= \mathbf{child} \mid \mathbf{descendant} \mid \mathbf{parent} \mid \mathbf{ancestor} \\
T &::= \mathbf{text} \mid \mathbf{node} \mid q \mid *
\end{aligned}$$

Here we assume an infinite supply of variables x, and recall that q is the metavariable of choice for node names. This language should look reasonable to anybody familiar with XQuery (or its update-enabled cousin XQuery! [11]).

The semantics of this language is given via a relation of the form

$$\Gamma \vdash \eta; e \Rightarrow \eta'; \bar{\ell},$$

where Γ is an environment mapping variables to values, η and η' are store histories, e is an expression, $\bar{\ell}$ is a value, and values are simply node sequences. Furthermore, it is an invariant of this relation that η' is an *extension* of η, that is, its base tree is identical, and the edit sequence of η is a prefix of the edit sequence of η'. The semantics has a few key properties:

- It is syntax-directed; Γ, η, and e are inputs and η' and $\bar{\ell}$ are outputs.
- The evaluation order is carefully specified.
- Update operations take effect immediately; queries evaluated after updates see the updated data.

We present only two of the evaluation rules – to give their flavor. The expression sequencing rule highlights the evaluation ordering; we will also demonstrate an update rule to show how these extend the store history. Expression sequencing is quite simple: we evaluate the first expression, perform any resulting updates, then evaluate the second expression in the updated tree:

$$\frac{\Gamma \vdash \eta_0; e_1 \Rightarrow \eta_1; \bar{\ell} \qquad \Gamma \vdash \eta_1; e_2 \Rightarrow \eta_2; \bar{\ell}'}{\Gamma \vdash \eta_0; e_1, e_2 \Rightarrow \eta_2; \bar{\ell}, \bar{\ell}'}$$

Here, $\bar{\ell}, \bar{\ell}'$ is the concatenation of the two sequences. The **delete** operation behaves this way:

$$\frac{\Gamma \vdash \eta_0; e \Rightarrow \eta_1; \bar{\ell}}{\Gamma \vdash \eta_0; \mathbf{delete}\ e \Rightarrow \eta_1, \mathbf{delete}_e(\bar{\ell}); \langle\rangle}$$

That is, **delete** evolves by first evaluating its argument to a sequence of nodes, then disconnecting all of those nodes. The remaining rules are similar; the one additional thing worth mentioning is that rules that result in new nodes pick fresh labels for them.

4.3 Path analysis

The static analysis of edits written in this language will proceed by identifying what *paths* the edit affects. We define paths as follows:

$$p ::= \varepsilon \mid loc \mid p|p \mid p/A :: T$$

These paths are given the standard meaning by a selection function $[\![p]\!]_d$ parameterized by the data that the path is indexing. (We'll skip the definition of this function.) We will say $p \subset p'$ when $[\![p]\!]_d \subset [\![p']\!]_d$ for all d. We also define a prefix function:

$$\begin{aligned} \mathit{pref}(\varepsilon) &= \varepsilon \\ \mathit{pref}(loc) &= \varepsilon \\ \mathit{pref}(p|q) &= \mathit{pref}(p)|\mathit{pref}(q) \\ \mathit{pref}(p/a :: t) &= p \end{aligned}$$

A path is prefix-closed when $\mathit{pref}(p) \subset p$.

The goal now is to analyze a high-level edit, identifying paths that it accesses and updates. Actually, we will need to settle for something a bit weaker: the existence of a conditional means that we cannot in general get this exactly right without referring to the data. So our static analysis will be somewhat conservative, in that it may report more accesses and updates than would actually happen on any data. At worst, this will cause us to conclude that some edits do not commute, even when they do. Thus, we may miss a potential optimization opportunity – but we will not have to sacrifice correctness!

Our analysis will be in the form of a syntax-directed inductive relation of the following form:

$$\Delta \vdash e \Rightarrow p_r; p_a; p_u$$

Here, Δ is an environment (much like Γ in the dynamic semantics) mapping variable names to paths. The inputs are Δ and e, a high-level edit, and the outputs are the three paths p_r of returned nodes, p_a of accessed nodes, and p_u of updated nodes. It is an invariant of this relation that p_a is prefix-closed; to maintain this invariant, we will define the prefix-closure function $\mathit{pref}^*(p)$:

$$\mathit{pref}^*(p) = \begin{cases} p & p \text{ is prefix-closed} \\ \mathit{pref}^*(p|\mathit{pref}(p)) & \text{otherwise} \end{cases}$$

It is fairly easy to show that this is well-defined (that is, that it terminates).

Many of the rules are straightforward, simply merging the accessed and updated paths from each subterm:

$$\frac{(x \mapsto p) \in \Delta}{\Delta \vdash x \Rightarrow p; \varepsilon; \varepsilon}$$

$$\frac{\Delta \vdash e \Rightarrow p_r; p_a; p_u \quad \Delta \vdash e' \Rightarrow p'_r; p'_a; p'_u}{\Delta \vdash e, e' \Rightarrow p_r|p'_r; p_a|p'_a; p_u|p'_u}$$

$$\frac{\Delta \vdash e \Rightarrow p_r; p_a; p_u \quad \Delta \vdash e' \Rightarrow p'_r; p'_a; p'_u}{\Delta \vdash \textbf{element}\ \mathit{loc}\ \{e\}\ \{e'\} \Rightarrow \mathit{loc}; p_a|p'_a; p_u|p'_u}$$

$$\frac{\Delta \vdash e \Rightarrow p_r; p_a; p_u \quad \Delta, x \mapsto p_r \vdash e' \Rightarrow p'_r; p'_a; p'_u}{\Delta \vdash \textbf{let}\ x := e\ \textbf{in}\ e' \Rightarrow p'_r; p_a|p'_a; p_u|p'_u}$$

$$\frac{\Delta \vdash e \Rightarrow p_r; p_a; p_u}{\Delta \vdash e/a :: t \Rightarrow p_r/a :: t; \mathit{pref}^*(p_r/a :: t)|p_a; p_u}$$

Others deserve a bit more discussion. The rule for **for** in fact only analyzes the body of the iteration once. This works out because the binding in Δ is bound to a path specifying many nodes; any uses of the variable in the body of the loop will be instantiated with a path referring to all the nodes that would be iterated over.

$$\frac{\Delta \vdash e \Rightarrow p_r; p_a; p_u \quad \Delta, x \mapsto p_r \vdash e' \Rightarrow p'_r; p'_a; p'_u}{\Delta \vdash \textbf{for}\ x\ \textbf{in}\ e\ \textbf{return}\ e' \Rightarrow p'_r; p_a|p'_a; p_u|p'_u}$$

The rule for **if** brings us to the main point of approximation involved in the analysis. The static analysis has no way of telling in general which branch of the conditional we will take. To handle this, we simply report any paths accessed or updated by *either* branch.

$$\frac{\begin{array}{c}\Delta \vdash e \Rightarrow r; a; u \\ \Delta \vdash e_t \Rightarrow r_t; a_t; u_t \\ \Delta \vdash e_f \Rightarrow r_f; a_f; u_f\end{array}}{\Delta \vdash \textbf{if } e \textbf{ then } e_t \textbf{ else } e_f \Rightarrow r|r_t|r_f; a|a_t|a_f; u|u_t|u_f}$$

Since deleting a node affects the semantics of all paths for which that node is a prefix, we must add all descendants to the list of updated paths when doing a **delete**. A similar comment applies to insertions.

$$\frac{\Delta \vdash e \Rightarrow p_r; p_a; p_u}{\Delta \vdash \textbf{delete } e \Rightarrow \varepsilon; p_a; p_u|p_r|p_r/\textbf{descendant} :: *}$$

$$\frac{\begin{array}{c}\Delta \vdash e \Rightarrow p_r; p_a; p_u \\ \Delta \vdash e' \Rightarrow p'_r; p'_a; p'_u\end{array}}{\begin{array}{c}\Delta \vdash \textbf{insert } e \textbf{ into } e' \Rightarrow \\ \varepsilon; p_a|p'_a|pref^*(p_r/\textbf{descendant} :: *); p_u|p'_u|p'_r|p'_r/\textbf{descendant} :: *\end{array}}$$

4.4 Commutativity

Before defining commutativity and stating the theorem that we can prove about it, we make one additional observation about the path analysis from above. Because paths rely critically on exactly what edges are available, we must be wary of anything that can delete edges. To ensure that paths generated by the analysis can reach the nodes that they identified at the time they were created, we should make sure that all the edges available at path-creation time are still available at path-evaluation time. We therefore define a conservative application function **capply**, which behaves much like **apply** on all but $\textbf{delete}_e$ edits:

$$\textbf{capply}(e, d) = \begin{cases} d & \exists E.e = \textbf{delete}_e(E) \\ \textbf{apply}(e, d) & \text{otherwise} \end{cases}$$

Then we will define containment for paths and path environments.

Definition 8. *We will say $\bar{\ell} \subset_\eta p$ (pronounced "p contains $\bar{\ell}$ in history η") when $\bar{\ell} \subset [\![p]\!]_{\textbf{capply}(\eta)}$. Similarly, $\Gamma \subset_\eta \Delta$ if $\Gamma(x) \subset_\eta \Delta(x)$ for all x.*

Finally, we can define commutativity. We will say two expressions commute when we can apply them in either order, resulting in identically updated trees and returning node sequences that differ only in order.

Definition 9. *We write $e \leftrightarrow_\Delta e'$ if for all η and Γ such that $\Gamma \subset_\eta \Delta$, we have*

$$\frac{\begin{array}{c}\Gamma \vdash \eta; e, e' \Rightarrow \eta_1; \bar{\ell} \\ \Gamma \vdash \eta; e', e \Rightarrow \eta_2; \bar{\ell}'\end{array}}{\mathbf{capply}(\eta_1) = \mathbf{capply}(\eta_2) \wedge bag(\bar{\ell}) = bag(\bar{\ell}')}$$

We can now state the main theorem:

Theorem 1. *Suppose that all of the following are true:*

$$\Delta \vdash e \Rightarrow p_r; p_a; p_u$$

$$\Delta \vdash e' \Rightarrow p'_r; p'_a; p'_u$$

$$p_u \# p'_a$$

$$p'_u \# p_a$$

$$p_u \# p'_u$$

Then $e \leftrightarrow_\Delta e'$.

4.5 Discussion

It is clear that the high-level language discussed here was designed for human production (and consumption). For that purpose, it has some very nice properties. The approach seems fairly flexible regarding exactly what operations are available in the high-level language and the semantics of these operations; for example, switching to a snapshot semantics should be possible with a bit of extra work. Moreover, the analysis itself should be simple enough to implement quickly (even though the techniques required to prove the desired properties get a bit hairy). All told, this approach seems much better suited to databases than the previous two on all the criteria.

There are some lessons to be learned here for revision control systems and text editors, as well. It probably is not straightforward to roll back changes or compute minimal edits in this language–at the very least, these problems were not discussed here–but there are other attractive features. The commutativity analysis discussed here may be useful in revision control systems for reconciling changes made in parallel; some analysis of this form, or perhaps greatly simplified, that guarantees that edits can be swapped regardless of the data they operate on would be quite helpful in that domain. Similarly, there may be something to be said about relocating changes within a text editor's edit history.

5 Conclusion

We have now seen a variety of approaches to the problem of representing edits. These have ranged from the very low-level, free-form, set-based foundations to high-level, expressive query and update languages. In the introduction, we

	patch triples	moves/copies	XQuery-like
rolling back	yes	yes	no
detecting changes	no	yes	no
reconciling parallel changes	partial	no	maybe
tracking provenance	yes	no	maybe
rolling back (again)	yes	yes	no
human readability	no	partial	yes
relocating changes	partial	maybe	maybe
human writability	no	no	yes
compact representation	no	yes	yes
provisions for concurrency	no	no	partial

Figure 6: Success of the three approaches

suggested some possible success criteria for these languages; Figure 6 summarizes the comments above with regards to these criteria. Cells marked "partial" indicate that the approach is not ideal for that criterion, but that it at least gets part way to achieving the goal associated with that criterion. Cells with "maybe", on the other hand, indicate that the success of that approach on that criterion are unclear, and that some further thought would be needed to clarify. None of the approaches are a perfect match for any of the applications under consideration.

For the domain of revision control, the best approach seems to be to take patch triples and extend it with some of the lessons learned from the change detection paper. Even doing so, there is more to be said in the realm of reconciling parallel changes by different authors. The direction forward likely involves some careful thought about adding facts that can let us view a repository as more structured than just a set. (For example, modeling Subversion might involve giving a linear ordering for the patches, and modeling git might involve attaching the patches to nodes of a DAG.) For text editors, the low-level language of Section 3 seems like a reasonable baseline to begin extending. An investigation of how exactly to support change relocation would make it a long way towards the needs of a text editor. Finally, the XQuery-like language seems to be the best fit of the three we examined for XML-backed databases. (This should be no surprise, recalling that the other languages were designed for significantly different applications.)

References

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

[2] T. Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Transactions on Computer-Human Interaction*

(TOCHI), 1(3):269–294, 1994.

[3] S. Boag, D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML query language. *W3C working draft*, 15, 2002.

[4] A. Cass and C. Fernandes. Using task models for cascading selective undo. *Task Models and Diagrams for Users Interface Design*, pages 186–201, 2007.

[5] S. Chawathe and H. Garcia-Molina. An expressive model for comparing tree-structured data. Technical report, Stanford University Database Group, 1997. Available at `http://www-db.stanford.edu/`.

[6] S.S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. *ACM SIGMOD Record*, 26(2):26–37, 1997.

[7] S.S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 493–504. ACM, 1996.

[8] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *ICDE*, page 0041. Published by the IEEE Computer Society, 2002.

[9] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[10] A. Dix. Moving between contexts. In *Design, Specification and Verification of Interactive Systems*, volume 95, pages 149–173. Citeseer.

[11] G. Ghelli, C. Re, and J. Siméon. XQuery!: An XML query language with side effects. *Current Trends in Database Technology–EDBT 2006*, pages 178–191, 2006.

[12] G. Ghelli, K. Rose, and J. Siméon. Commutativity analysis in XML update languages. *Database Theory–ICDT 2007*, pages 374–388, 2006.

[13] J.W. Hunt, M.D. McIlroy, and Bell Telephone Laboratories. *An algorithm for differential file comparison*. Bell Laboratories, 1976.

[14] D. Kurlander and S. Feiner. A visual language for browsing, undoing, and redoing graphical interface commands. *Visual languages and visual programming*, pages 257–275, 1990.

[15] Andres Löh, Wouter Swierstra, and Daan Leijen. A Principled Approach to Version Control.

[16] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, page 706. ACM, 2006.

[17] J. Minker. Foundations of deductive databases and logic programming. 1987.

[18] G. Sur, J. Hammer, and J. Siméon. An XQuery-based language for processing updates in XML. *Informal Proc. PLAN-X*, 2004, 2004.

[19] Y. Wang, D.J. DeWitt, and J.Y. Cai. X-Diff: An effective change detection algorithm for XML documents. In *Proceedings of the 19th International Conference on Data Engineering*, pages 519–530. IEEE, 2004.