

A Version Numbering Scheme with a Useful Lexicographical Order*

Arthur M. Keller[†]
Stanford University
Computer Science Dept.
Stanford, CA 94305-2140

Jeffrey D. Ullman[‡]
Stanford University
Computer Science Dept.
Stanford, CA 94305-2140

Abstract

We describe a numbering scheme for versions with alternatives that has a useful lexicographical ordering. The version hierarchy is a tree. By inspection of the version numbers, we can easily determine whether one version is an ancestor of another. If so, we can determine the version sequence between these two versions. If not, we can determine the most recent common ancestor to these two versions (i.e., the least upper bound, lub). Sorting the version numbers lexicographically results in a version being followed by all descendants and preceded by all its ancestors. We use a representation of nonnegative integers that is self delimiting and whose lexicographical ordering matches the ordering by value.

1 Introduction

Our motivation is a project in collaborative design for Civil Engineering being carried out at Stanford. The approach taken by the project, named CEDB (Collaborative Environment for the Design of Buildings), leads to an interesting problem in version numbering. Our approach to version numbering is the subject of this paper.

CEDB uses a version and configuration management system that supports incremental checking of constraints (see Krishnamurthy [1993]). There is a hierarchical version system, with a separate version hierarchy for each design discipline (e.g., plumbing, electrical, structural engineering). A *configuration* consists of a version from each discipline together with a set of constraints that the configuration should satisfy (although we do not insist on constraint satisfaction in

all situations). A configuration C_1 may have an ancestor configuration C_0 , in which each of the versions participating in configuration C_1 must be a descendant of the corresponding version in configuration C_0 . The CEDB computational strategy is to check incrementally for constraint violations by comparing a configuration with an ancestor configuration. This comparison relies on the determination of changes (“deltas”) that occurred between two versions.

Our approach is to record the incremental changes between a version and its immediate ancestor. As a result, if we need to find the changes that have occurred between versions A and B , we must find the changes associated with all versions that lie between A and B in the version hierarchy. Imagine that all changes are stored in a relation, along with the version to which they pertain. For example, if in going from version P to a child version C , we insert the element a , then we might store the *change record*, a triple (`insert`, a , C) in the relation.

Ideally, we should be able, given version identifiers A and B , to find all the change records associated with versions that lie between the version numbers for A and B in the version hierarchy (including B but not A). Surely, we can do so if we have access to the hierarchy. However, there is a lack of efficiency in an approach that issues one query for each version on the path between A and B . We propose instead to issue one range query that will access all the change records for all the versions between A and B . With the proper index and storage structure, these change records can be obtained in little more than the time it takes to access data of this bulk.

Our primary goal, therefore, is to find a scheme for numbering versions, so that all of the relevant change records can be found by a sequential scan of the data between the two version numbers when change records are sorted by their associated version number. There are two interesting aspects of our version numbering scheme.

*This work is part of the CEDB project: Collaborative Environment for the Design of Buildings, or Civil Engineering Database. This effort is funded in part by NSF grant IRI-91-16646.

[†]Arthur Keller's e-mail address is `ark@cs.stanford.edu`.

[‡]Jeff Ullman's e-mail address is `ullman@cs.stanford.edu`.

1. A method for assigning numbers to versions so that all the versions between A and B in the hierarchy have version numbers that lie between the version numbers for A and B (however, there may also be other version numbers that lie between A and B). The version numbering scheme must make it possible to add new children without renumbering old versions.
2. A method for encoding nonnegative integers as character strings so that if $i < j$, then the code for integer i is lexicographically less than the code for j .

Note that the conventional encoding of integers does not satisfy (2). For example, in decimal the character strings for the integers 5 and 43 are in the wrong order. That is, as integers, $5 < 43$, but as character strings, **43** lexically precedes **5**.

2 Representing a Hierarchy

In this section, we describe how to represent the hierarchical relationship of version numbers using a linear encoding that sorts in a useful manner. Let us consider the simple version hierarchy in Figure 1, where the names of the versions offer no help in finding the versions that lie between two given versions in the hierarchy.

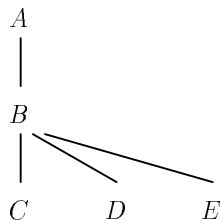


Figure 1: Simple version hierarchy.

Version A has one child, version B , which in turn has three children, versions C , D , and E . In this case, the lexicographical ordering of the version names is useful. To trace the ancestry of version D , we can progress sequentially through the ordering. We traverse, A , B , skip C , and then stop when we reach D .

Observe that it is not possible to order the versions so that a sequential traversal of the ordering will encounter all the ancestors of each version in an uninterrupted sequence. (For example, consider a parent has three children. Given some sequential ordering of

the parent and the three children, a sequential traversal from at least one of the children must skip over at least one of its siblings to reach its parent.) In general, we shall have to skip some versions, but it is highly desirable to find all the ancestors of a version within some small region of the complete ordering. For example:

- If we wish to find the changes from some version V to a descendant version W , we shall need to retrieve less information from secondary storage if the ancestors of W are in a small region of the ordering.
- If we store all change records in a single relation, and we sort or index those records by version number, we can use a range query to help find the changes between a version V and a descendant version W , and this query will be efficiently implemented.

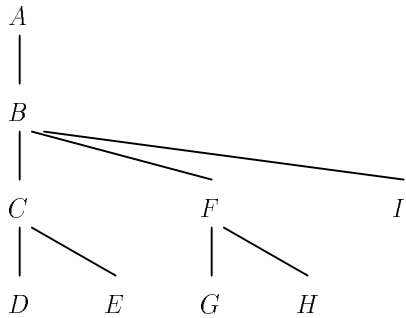
Now suppose we wish to add to the hierarchy. We assume that new versions can only be added as children of existing versions, as is the case in the CEDB system. When we add new versions, we are faced with the problem of finding new names for the new versions, especially if we wish to retain the property that names are in lexicographic order along paths of the hierarchy.

Figure 2 shows two possible approaches. In each case, we have added some children to Figure 1. In Figure 2(a) we have renamed versions so children follow their parent in lexicographic order. Figure 2(b) shows another approach, where names of versions are unchanged and new versions are given names that follow any previously used names in lexicographic order.

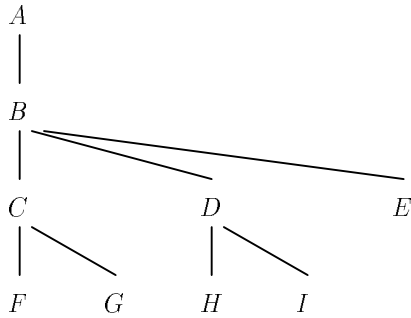
The primary advantage of ordering new versions near the parent as in Figure 2(a) is locality of reference. The primary advantage of naming versions in order of creation as in Figure 2(b) is that we do not have to rename versions when adding other versions. However, in the latter scheme there tend to be many “false drops,” that is, version names that lie in lexicographic order between the names of some version V and a descendant W , yet are not located between V and W in the hierarchy.

By using variable-length version names, we can avoid the problem of renaming. Consider Figure 3, where we have encoded the ancestry in the version name. That is, the name of each version is the name of its parent, followed by a number that indicates which child it is in the ordering of children of its parent.

There are several further improvements we can make to this scheme. First, we can optimize for the common case where a version has only one child. That



(a) Sorted with parents.



(b) Sorted by level.

Figure 2: Sorting versions.

is, we may assign to the first child of certain versions a number that is one greater than the number of its parent, than treating all *alternatives* (children of the same parent version) the same, as shown in Figure 4. This approach reduces the average length of names in the common case when most versions have one child. In comparison, the scheme of Figure 3 forces names to be as long as the paths in the hierarchy.

In summary, there are three important criteria we need from a version-naming scheme.

1. The *lexicographic property*: all the versions between some version V and a descendant version W lie lexicographically between V and W .
2. Given any hierarchy of versions, we can find a name for any additional child of an existing version. This name must be distinct from the names of other versions in the hierarchy.
3. The length of a name does not grow significantly along paths that follow a sequence of leftmost children.

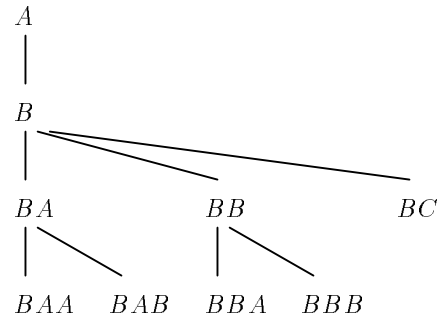


Figure 3: Encoding ancestry.

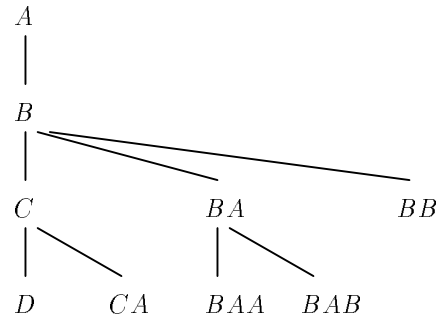


Figure 4: Another way to encode ancestry.

3 Our Proposed Version-Numbering Scheme

Note the problem in naming the descendants of version BA of Figure 4. We need to distinguish between the first child of BA and the alternative to BA , both of which could plausibly have been named BB . To handle this problem, we shall alternate between encoding the generation number and encoding the alternative number. To facilitate the exposition, we will use numbers to represent the generation sequence and letters to represent alternatives. However, in version names, think of A, B, \dots as representing integers $0, 1, \dots$. This approach is illustrated in Figure 5.

Formally, a version number for a version V is a sequence $g_0 a_1 g_1 \dots a_n g_n$. Here, g_n indicates the number of generations from the *matriarch* of V , whose number is $g_0 a_1 g_1 \dots a_n 0$. The matriarch of V is the closest (not necessarily proper) ancestor of V that is *not* a first child of its parent (or the root if there is no such ancestor of V). For example, in Figure 5, the matriarch of $1B1A2$ is $1B1A0$, and the matriarch of $3A2A0$ is itself.

The number a_n is one less than the number of siblings that the matriarch of V has to its left. For exam-

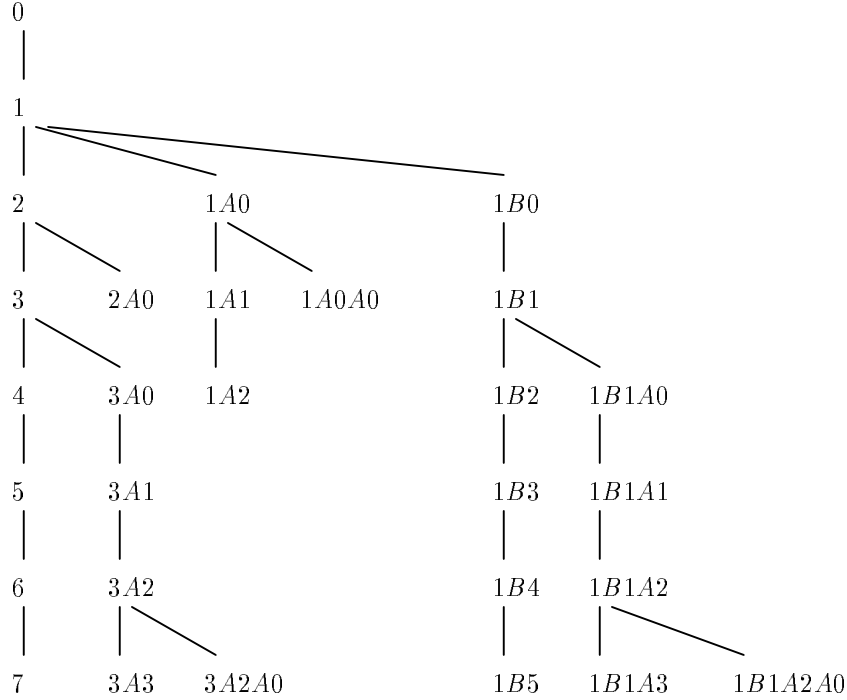


Figure 5: Improved version numbering.

ple, in Figure 5 consider the children of version 1. The first child gets number 2. The second child instead has appended to the name “1” the letter *A* (which represents “0”). The third child has *B* appended to the name “1”. Both the second and third children have a 0 appended to their names after the *A* or *B*, because they are their own matriarchs; i.e., they are 0 generations from their matriarch. That is, the following rules summarize how the children of a version are named.

- The first child of version $g_0a_1g_1 \dots a_n g_n$ is $g_0a_1g_1 \dots a_n(g_n + 1)$.
- The k th child of version $g_0a_1g_1 \dots a_n g_n$, for $k > 1$, is $g_0a_1g_1 \dots a_n g_n(k - 2)0$.

This scheme offers an important economy in the situation that is most common: very few versions have more than one child. For example, in the extreme case where there are n versions in a chain, that is, no version has more than one child, $\log n$ bits suffice to represent version names. In comparison, any scheme that allows us to extend the name of a version to get the name of its child would require linear space to represent these n version names.

4 Some Basic Algorithms Using the Version-Numbering Scheme

We begin with an algorithm for determining the ancestors of a version.

Algorithm 1 (Traverse ancestors)

Input: Version $g_0a_1g_1 \dots a_n g_n$.

Output: All ancestors in order from parent to oldest.

```

for  $i := n$  downto 0 do
  for  $j := g_i$  downto 0 do
    Print ( $g_0a_1g_1 \dots g_{i-1}a_i j$ )
  
```

◇

Note that when i is 0, then merely j is output.

Here is an algorithm for determining the closest common ancestor (i.e., least upper bound) to two versions.

Algorithm 2 (Closest common ancestor)

Input: Versions $g'_0a'_1g'_1 \dots a'_n g'_n$ and $g''_0a''_1g''_1 \dots a''_m g''_m$.

Output: Closest common ancestor $g_0a_1g_1 \dots a_l g_l$.

```

if  $g'_0 \neq g''_0$ 
  then begin  $l := 0; g_0 := \min(g'_0, g''_0)$ ; exit end
  else  $g_0 := g'_0$ ;
for  $i := 1$  to  $\min(n, m)$  do begin
  if  $a'_i \neq a''_i$ 
    then exit;
   $a_i := a'_i$ ;
  if  $g'_i \neq g''_i$ 
    then begin  $l := i; g_i := \min(g'_i, g''_i)$ ; exit end;
   $g_i := g'_i$ ;
  end;
 $l := \min(n, m)$ ;
exit
◇

```

We call the traversal that results from lexicographical ordering from our encoding the *inverse left-corner traversal*. This traversal is produced by the following algorithm.

Algorithm 3 (Inverse left-corner traversal)

Input: A version history represented by tree T .

Output: Traversal according to lexicographical ordering.

```

walk(T) =
  for  $i := 2$  to last child of root do
    walk( $i$ th subtree of root of  $T$ );
  print root of  $T$ ;
  walk(first subtree of root of  $T$ )
end;
◇

```

To traverse from a version v_a to some descendant v_d , scan the ordering starting at the version v_a and ending at the desired descendant v_d , skipping all versions that are not ancestors of the desired descendant. The only versions that will have to be skipped are other descendants of ancestor v_a .

5 Representing Version Names

We may see the version naming scheme of Section 3 as one where version names are lists of integers, the alternating g 's (generation numbers) and a 's (position among siblings). These lists have a natural lexicographic order. That is, to compare two lists, find the common prefix and delete it. After this modification, an empty list precedes any list, and otherwise, the list with the lower first element precedes the other. For example, in Figure 5, $1B1A2$ precedes both $1B4$ and $1B1A2A0$.

It is easy to check that with this numbering scheme, all the versions between a version V and its descendant

W lie between V and W in lexicographic order. In a sense, that is all we need. However, the scheme of Section 3 suffers from the problem that the elements of the lists that form names are arbitrary integers. Since we must represent such integers in some system that uses a finite number of symbols, we may not be able to preserve the lexicographic order.

For example, consider the names $(10)A0$ and $2B3$. The first of these represents a version that is found by following 10 generations of first children from the root, then moving to the second child. The second is found by following two generations of first children from the root, moving to the third child, and then going down three more generations of first children. The latter precedes the former, since $2 < 10$. However, if we represent integers in decimal, and treat A as 0, B as 1, and so on, then the two names become 1000 and 213 . The first precedes the second in lexicographic order, which is wrong.

We thus face the problem of representing integers by codes that are in the same order, lexicographically, as the values of the integers. Such an encoding of integers will be said to have the *lexicographic property*. We just observed that the ordinary representations such as decimal do not have the lexicographic property. We saw that $2 < 10$ as integers, but 10 precedes 2 lexicographically.

The simplest solution is to use fixed-length numbers of sufficient length, with numbers padded on the left by leading zeroes. For example, 5 digits are enough to handle 10,000 generations of versions and up to 10,000 children of one version. The fixed-length solution suffers from three problems: wasted space for low values, limited maximum value, and the need to specify the length somehow.

Alternatively, we can use a variable-length representation that is self-delimiting. Consider the following simple scheme for representing binary numbers. Represent a binary number of b bits by prefixing it with $b-1$ one bits and a zero bit. This scheme is called Elias codes [Elias 1975]. For example, the number 5, or 101_2 , is represented by the binary string 110101. Decoding is easy. The prefix consists of all the bits up to and including the first zero; the number of bits in the prefix tells how many bits follow. The number follows in binary.

A similar scheme can be used for variable-length encodings using decimal numbers. Here, h digits with the value 9 are followed by $h+1$ digits, where the first of those latter digits is not 9. The numbers 0 through 8 are represented as themselves. The numbers 9 through 98 are represented by 900 through 989 . The numbers 99 through 998 are represented by 99000 to

99899, and so on. In general, to get the value from such a representation add the prefix (the consecutive 9's) to the rest of the number using ordinary decimal arithmetic.

To obtain a representation we use the following algorithm. Let d be the number of digits ordinarily used to represent the desired value v . If all d digits of v are the digit 9, then the representation is d 9's followed by $d + 1$ 0's. Otherwise, let $v' = v - x$, where x is the decimal number formed by $d - 1$ 9's; that is, $x = 10^{d-1} - 1$. Then the representation of v is $d - 1$ 9's followed by v' padded out with leading zeroes to d digits, if necessary.

We might also adapt the same idea to any radix other than binary or decimal. In any radix it is easy to prove that for any integers i and j , if $i < j$, then the encoding of i precedes the encoding of j , lexicographically. We shall focus on the binary system, for simplicity.

Suppose $i < j$. If i in binary has fewer bits than j , then the encoding of i will begin with fewer 1's than j 's encoding. Thus, i precedes j lexicographically, when both are encoded. The only other possibility when $i < j$ is that i and j have the same number of bits in their binary representations, say b bits. Then both their encodings start with $1^{b-1}0$, followed by the b -bit representations of i and j . Since $i < j$, we see that the encoding of i will precede that of j , lexicographically.

6 Optimal Lexicographic Encodings of Integers

Let us define the *expansion* of an encoding to be the limit, as i gets large, of the ratio of the length of the encoding of i to the length of the ordinary radix representation of i in the base that has as many digits as the encoding uses. For example, whether in binary, decimal, or another base, the expansion of the encoding schemes of Section 5 is 2. The reason is that the prefixes introduced have, within 1, as many digits as the ordinary representation of the number.

We might ask whether 2 is the minimum expansion possible. The answer is rather surprising. It is possible to approach expansion 1 as closely as we like; that is, asymptotically the requirement of a lexicographic encoding costs nothing. However, to approach 1 it is necessary to indefinitely increase the base of the representation. That has the effect of putting a progressively higher floor on the minimum length of an encoded integer when all representations are reduced to binary, as they surely must be for representation within a computer.

We shall introduce the optimal scheme first by considering the case of the decimal representation.

Encode the first 5 numbers (0 through 4) by 0 through 4.

Encode the next 25 numbers (5 through 29) by 50 through 74.

Encode the next 125 numbers (30 through 154) by 750 through 874.

Encode the next 625 numbers (155 through 779) by 8750 through 9374.

Encode the next 3125 numbers (780 through 3904) by 93750 through 96874, and so on.

This approach offers smaller expansion of codes at the expense of slightly more complicated encoding and decoding. For example, the scheme of Section 5 represents 999 numbers in 5 digits, while this scheme represents 3905 numbers in 5 digits. Here are algorithms for encoding and decoding numbers using this scheme.

Algorithm 4 (Encoding)

Input: n to be encoded.

Output: Decimal representation of the encoded number.

Let d be the smallest integer such that $n < (5^d - 5)/4$. Let $.h_1h_2 \dots h_{d-2}$ be the decimal fraction representation of $1 - 1/2^{d-2}$.

Then the encoding of n is the decimal representation of $h_1h_2 \dots h_{d-2}0$, plus n , minus $(5^{d-1} - 5)/4$.

◇

For example, let $n = 2000$. Then $d = 6$, because $(5^6 - 5)/4 = 3905$ is greater than 2000, but $(5^5 - 5)/4 = 780$ is not. Thus, $h_1h_2 \dots h_{d-2} = 9375$, since $1 - \frac{1}{16} = .9375$. We thus compute the encoding of 2000 by adding 93,750 to 2000 and subtracting $(5^5 - 5)/4$, or 780. The resulting code is 94970.

Algorithm 5 (Decoding)

Input: A k -digit encoding, whose decimal value is d , to be decoded.

Output: Numerical value of the decoded number.

The encoded integer is $d - 10^k + 2 \times 5^k + (5^k - 5)/4$.

◇

For example, suppose we are given the code 94970. Then $d = 94,970$ and $k = 5$. We must subtract $10^5 = 100,000$ from d , then add $2 \times 5^5 = 6250$ and add $(5^5 - 5)/4 = 780$. The result is 2000.

7 Optimality of Our Integer Encoding

It turns out that the encoding of Section 6, or its generalization to an arbitrary base, is essentially the best we can do. We shall show that any encoding in base b that satisfies two constraints appears to use only half of the b available digits. The two constraints, both of which are consequences of our assumptions about how version names are to be encoded, are:

1. The encoding must be a *prefix* code; that is, no integer may have an encoding that is a prefix of another integer's encoding. To understand the need for this requirement, we must remember that whatever alphabet is used to represent encodings is the *entire* alphabet. There is no extra "blank" or "endmarker" symbol available. Thus, if one integer's encoding were a prefix of another's, it would not be clear which integer was meant when the latter's appeared.
2. The encoding must not be biased in favor of large or small integers. The formal requirement that we believe reflects this condition is that the number of encodings of a given length i grows in a uniformly exponential way. That is, the number of integers that have encodings of length i is c^i for some constant c . (While it is possible to develop codes that favor either small or large numbers, we favor the common case where version numbers may be arbitrarily large.)

Condition (1) has a well-known arithmetic interpretation, called the *Kraft inequality* (McMillan [1956], Cover and Thomas [1991]). Let n_i be the number of integers with codes of length i , and let b be the number of symbols in the encoding alphabet. Then if the encoding satisfies condition (1), i.e., it is a prefix code, then

$$\sum_{i=1}^{\infty} n_i b^{-i} \leq 1$$

Now, we may use condition (2) to say that $n_i = c^i$ for some constant c . That, combined with the Kraft inequality says

$$\sum_{i=1}^{\infty} (c/b)^i \leq 1$$

If we sum the infinite series we get

$$\frac{c/b}{1 - (c/b)} \leq 1$$

or $c \leq b/2$. That is, the number of integers that can be assigned codes of length i in an encoding using b

symbols is $(b/2)^i$. Note that the scheme of Section 6, which uses $b = 10$, actually assigns exactly this many integers.

For a general base b , we can represent b^n numbers by codes of length up to m , provided m satisfies

$$(b/2)^m + (b/2)^{m-1} + \dots + 1 \geq b^n$$

Some algebra shows that the smallest m that satisfies the above is approximately

$$m = n \log_2 b / (\log_2 b - 1)$$

Put another way, the expansion of such a code is approximately $1 + 1/\log_2 b$. For $b = 10$, we have an expansion of about 1.3, compared with an expansion of 2 for the encoding of Section 5.

Of course, we need to show the existence of an encoding for an arbitrary number of symbols b that preserves lexicographic order the way the encoding for $b = 10$ does. We claim that such a code always exists. The existence is easy to exhibit and also comparatively easy to understand for the case that b is a power of 2; say $b = 2^k$. It also helps our understanding if we represent the digits in base b by their binary representation. For example, if $b = 16$, we represent each of the hexadecimal digits by a string of four 0's and 1's.

Then the code consists of the following:

2^{k-1} bit strings of length k beginning with 0,
 2^{2k-2} bit strings of length $2k$ beginning with 10,
 2^{3k-3} bit strings of length $3k$ beginning with 110,
 2^{4k-4} bit strings of length $4k$ beginning with 1110,
 2^{5k-5} bit strings of length $5k$ beginning with 11110,
and so on.

For example, again assuming $b = 16$, so $k = 4$, our code uses 8 bit strings of length 4, which are 0000 through 0111, corresponding to the hexadecimal digits 0 through 7. We have 64 bit strings of length 8, each of which begins with 10. These are the two-digit hexadecimal numbers whose first digit is between 8 and 11 and whose second digit is arbitrary. The last of the five listed cases above (strings of length $5k$) tells us that there are 32,768 bit strings of length 20, each beginning 11110. These correspond to hexadecimal numbers whose first digit is 15 and whose second digit is between 0 and 7; the remaining three digits are arbitrary.

If we assign codes to integers lowest first, it should be clear that the codes will have the lexicographic property. The reason is that shorter strings always precede longer ones in lexicographic order, because the longer the string the longer the initial prefix of 1's. Among strings of the same length, their lexicographic and numeric orders are the same.

The expansion of this encoding scheme is easily seen to approximate $1 + (1/k)$. Thus, by picking a higher power of two as the base, we can create a code with expansion arbitrarily close to 1. Moreover, we can represent the digits in base $2^k = b$ in binary, so we do not have a problem with representing arbitrarily large alphabets.

The only downside with making k as large as we like is that we do worse when we encode small integers. The reason is that no integer has an encoding with fewer than k bits. For example, if we choose the code with $b = 2^k$ and represent base- b digits in binary, then the integers 0 and 1 have codes k times as long as their binary representations, namely 0^k and $0^{k-1}1$, respectively.

Fortunately, there are reasonable compromises. For example, we can pick $k = 8$, thereby representing integers by byte strings, with a 12.5% expansion. Version names, which it should be recalled is what we really wish to represent, are then likewise represented by bit strings. Because of the prefix property of our encoding, we can represent a list of integers, which are the version names according to the scheme of Section 3, concatenate them, and still are able to pull the encoded byte string apart into its constituent encoded integers.

8 Related Work

Katz [1990] gives a survey of version modeling in engineering databases. Some of approaches he mentions allow the version hierarchy to be a DAG (directed acyclic graph), which we do not handle. However, there is no mention of version naming schemes in this survey. Ambriola et al., [1990] gives a survey of version control and configuration management in the software engineering environment. The SCCS system by Rochkind [1975] allows a version hierarchy, but version numbering is limited to release number (for major or important changes) before the decimal point, followed by version number (for minor changes). The deltas between a parent and child version are insertions and deletions to the parent version. All of the deltas for an entire version hierarchy are stored as change records (nested if necessary) in a single UNIX file. The deltas are thus sorted by where they apply in the version. To compute a version or the deltas between any two versions with an ancestor relationship, it is necessary to scan the entire file.

The RCS system by Tichy [1985] includes an explicit revision tree with some of the characteristics of our scheme, but less general and less compact. The version hierarchy is limited to four levels. The first

two levels are release and revision number, as in SCCS. When a new branch is created (i.e., a second child), then two extra numbers are added: the branch number followed by the child number of that branch. For example, suppose version 1.3 already has a child version 1.4 and wants to create a new child version. That new child version is numbered 1.3.1.1. A subsequent child of version 1.3 is numbered 1.3.2.1. The first child of version 1.3.1.1 is called 1.3.1.2. There is a problem with naming the second child of version 1.3.1.1. The approach used in RCS is to pick the next available branch number. Thus, the second child of version 1.3.1.1 is named 1.3.3.1.

Zobel, Moffat, and Sacks-Davis [1992] consider integer encoding for full-text database systems. They consider efficient one-pass encoding schemes, but are not concerned with the lexicographical ordering properties of those schemes. Bentley, Sleator, Tarjan, and Wei [1986] consider compression schemes with locality of reference. These are prefix codes, but they do not obey the property that the lexicographical ordering of the encoding matches the ordering of the numbers encoded. Rather, their objective is to use shorter encodings for frequently used values, like Huffman codes. Gallagher and Van Voorhis [1975] encode infinite sources (like our encoding of the non-negative integers) using a Huffman-like encoding. They also give shorter encodings to frequently used values, while we give shorter encodings to values that are smaller. We assume that smaller values are used more frequently because there are many version hierarchies that start with the version number zero, and fewer and fewer version numbers with larger and larger values.

9 Conclusion

We have presented a version numbering scheme for versions with alternatives that has a useful lexicographical ordering. The version numbering scheme uses two techniques. The first technique encodes the version hierarchy, that is, the part from the root to the given version. Our approach favors the common case of the leftmost child, so that the length of the version name for this child is at most slightly longer than its parent. A version's number is proportional in length to the number of times that one of its ancestors was not the leftmost child.

The second technique is to encode nonnegative integers so that their lexical order matches the ordering of their values. This encoding is also self-delimiting and optimal.

We are implementing these methods in the version and configuration management system being devel-

oped for the CEDB project.

10 Acknowledgments

We thank members of the CEDB project for their feedback. We thank Marianne Siroker for her assistance in preparing this paper.

11 Bibliography

- Ambriola, V., L. Bendix, and P. Ciancarini [1990]. “The evolution of configuration management and version control,” *Software Engineering Journal*, November 1990, pp. 303–310.
- Bentley, J. L., D. D. Sleator, R. E. Tarjan, and V. K. Wei [1986]. “A Locally Adaptive Data Compression Scheme,” *Communications of the ACM* **29**:4, April 1986, pp. 320–330.
- Cover, T. M. and J. A. Thomas [1991]. *Elements of Information Theory*, Wiley, New York.
- Elias, P. [1975]. “Universal Codeword Sets and Representations of the Integers,” *IEEE Trans. on Information Theory* **IT-21**, pp. 194–203.
- Gallagher, R. G. and D. C. Van Voorhis [1975]. “Optimal Source Codes for Geometrically Distributed Integer Alphabets,” *IEEE Trans. on Information Theory*, March 1975, pp. 228–230.
- Katz, R. [1990]. “Towards a unified framework for version modeling in engineering databases,” *ACM Computing Surveys*, pp. 375–408.
- Krishnamurthy, K. [1993]. “Version and configuration management for collaborative design,” TR-92, Stanford Center for Integrated Facility Engineering, Nov., 1993.
- McMillan, B. [1956]. “Two inequalities implied by unique decipherability,” *IEEE Trans. Information Th.* **IT-2**, pp. 115–116.
- Rochkind, M.J. [1975]. “The Source Code Control System,” *IEEE Trans. on Software Eng.*, **SE-1**:4, pp. 364–370.
- Tichy, W.F. [1985]. “RCS — A System for Version Control,” *Software—Practice and Experience* **15**:7, John Wiley, pp. 647–654.
- Zobel, J., A. Moffat, and R. Sacks-Davis [1992]. “An Efficient Indexing Technique for Full-Text Database Systems,” *18th VLDB*, 1992, pp. 352–362.