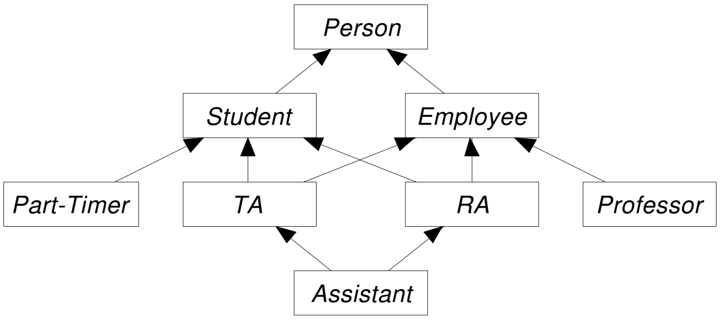


Figure 1: Partial university hierarchy.

# 1 Introduction

Multiple inheritance hierarchies arise in many areas, including knowledge representation and reasoning, database management, query processing, and programming. Neural network models, semantic networks, and state spaces all require such a structure. Complex-object databases, where objects are grouped into classes organized into an inheritance hierarchy, are becoming more prevalent. Database researchers have recognized the utility of hierarchies in querying databases. Scheduling packages and computer-aided design software such as UML (Unified Modeling Language) and OMT (Object Modeling Technique) employ hierarchal structures. Object-oriented programming languages such as C++ and Java are based on organizing classes into a hierarchy. All of these areas have one common requirement – a compact representation of the hierarchy with the ability to compute relationships efficiently.

In some cases, objects that are instances of classes organized in an inheritance hierarchy are manipulated via expressions involving conjunction, disjunction, and relationship testing, representing greatest lower bounds (GLB), least upper bounds (LUB), and subsumption. Embedding the hierarchy in a boolean lattice enables the expressions to be evaluated as binary AND, OR, and containment. Permitting multiple inheritance complicates the expressions involving conjunction and disjunction, leading to the possibility of the result representing a disjunction or conjunction of classes, respectively, as illustrated below.

Consider for example the inheritance hierarchy of Figure 1. The class *Person* is an immediate superclass of *Student*, while *Student* is an immediate subclass of *Person*. Class *TA* is an immediate subclass of both *Student* and *Employee*, and thus inherits all properties of both classes. Class *TA* is also a subclass of class *Person*.

With the assumption that an object is created in its most specialized class, the computation of the join of two classes is required. For example, the claim that an object is both a *TA* and an *RA* in the simple university hierarchy implies that the object is in the class representing the conjunction of the two classes, or in lattice terms, the GLB of the two classes, which is class *Assistant*. Similarly, the LUB operation represents the lowest superclass of an object if it is known that it is in one of several classes. For example, an object in either the *Student* or *Employee* class is known to be in the *Person* class.

| Classes | Ait-Kaci | Ganguly | Caseau | Agrawal | Fall |
|---|---|---|---|---|---|
| *Person* | 11111 | 11111 | 00000 | [1,8] | [] |
| *Student* | 11011 | 10111 | 00001 | [1,3], [4,5] | 1 |
| *Employee* | 11101 | 01111 | 00010 | [4,7], [2,2] | [1] |
| *Part-Timer* | 00010 | 10110 | 00101 | [1-1] | 2 |
| *TA* | 01001 | 10101 | 01011 | [2,2], [4,4] | 1.[2] |
| *RA* | 10001 | 01101 | 10011 | [4,5] | [1-1] |
| *Professor* | 00100 | 01110 | 00110 | [6,6] | [1-[1]] |
| *Assistant* | 00001 | 00001 | 11011 | [4,4] | 1.[2-1] |

Figure 2: Encodings of university hierarchy.

Multiple inheritance further complicates these operations as the result may not be a single class. In the example, the GLB of classes *Student* and *Employee* is the two classes *TA* and *RA*, representing the notion that a person who is both a student and an employee must be either a TA or RA, or both. Also, the LUB of classes *TA* and *RA* is the two classes *Student* and *Employee*, representing the notion that a person who is either a *TA* or *RA* is both a student and employee.

Previous representations of hierarchies have involved static, compile-time encodings which require recalculation for any updates. A dynamic approach is more appropriate as many applications contain persistent data and hierarchies are constantly changing. Recomputation of the encoding is not only too time consuming, but also requires reassigning new codes to objects in persistent applications – an expensive operation. The ideal encoding allows for run-time additions to the hierarchy to be efficiently encoded with little storage overhead and requires few changes to existing codes.

Four earlier methods are reviewed in Section 2, along with a fifth which fails to support GLB and LUB operations. Details of the algorithms can be found in the references, but are not necessary for the understanding of this paper. Variations of two of the methods are developed in Section 3, and experimental results presented in Section 4. Summary remarks follow in Section 5, along with directions for future research.

## 2 Background

Ait-Kaci et al. [AKBLN89] develop an encoding method based on a compressed version of transitive closure by only using new bit positions where necessary, as illustrated in Figure 2. Unfortunately the encoding is done from the bottom up; that is, the lowest levels of the hierarchy are encoded first. The method is also static, and uses a large compile-time overhead to create the encoding. This impedes the incremental update of the hierarchy and fails to model applications that are developed over time from the top down, as is the case with many persistent hierarchies. The encoding produced is compact relative to transitive closure in the sense that new bit positions are added only to differentiate nodes with common descendants. This produces an encoding with constant-time, very efficient calculations of GLB, LUB, and subsumption, using binary AND, binary OR, and comparison, respectively. A method termed *modulation* is employed to group a hierarchy into *modules*, and thus allow for more reuse of bit positions by adding a group code to a class. This produces a much more compact encoding for hierarchies that group well, and does not add much overhead to the lattice operations. Both encoding methods achieve a best case of $O(\log n)$ and worst case of $O(n)$ length codes where $n$ is the number of nodes in the hierarchy.

Ganguly et al. [GMR94] improve on these results by developing an algorithm which is shown via empirical experiments to be significantly more space efficient and does not rely on the structure of the hierarchy. Again, the algorithm is static, as it employs both bottom-up and top-down traversals of the hierarchy. Such a method requires a complex transformation to enable dynamic additions to the hierarchy. The computation of LUB and subsumption are similar to that of Ait-Kaci et al., but the computation of GLB requires the addition of a code representing the classes with multiple parents.

Caseau [Cas93] achieves a more compact binary encoding of a hierarchy which supports subsumption but does not allow GLB and LUB computations. A sample encoding is shown in Figure 2. His encoding first requires the transformation of the hierarchy into a lattice. The encoding then proceeds by locating the primary nodes (nodes with unique parents) and assigning a bit position to them. The choice of bit position for a node relies on the positions used for the nodes related to the node's ancestors. The algorithm works from the top down, and modifications to chosen bit positions are required if conflicts occur when a new node is added. Since GLB and LUB computations are not supported, bit positions are reused in nodes shown to be unrelated by other inherited positions. For example, classes *Part-Timer* and *Professor* share bit position three in the encoding in Figure 2. The overall encoding produced is much more compact than that of Ait-Kaci et al. because it does not support the lattice operations. The top-down nature of the algorithm is borrowed in the implementation of an inverted version of the encoding of Ait-Kaci et al. in Section 3.

Agrawal et al. [ABJ89] propose a range compression scheme that assigns an interval to a class (or multiple intervals to support multiple inheritance), as illustrated in Figure 2. A class is a subclass of another if its interval(s) fall into the interval range(s) of the superclass. Encoding is done by first performing a post-order traversal of a spanning tree for the hierarchy. Lattice operations can then be performed by comparing the intervals of the classes involved. Although creating a spanning tree for a hierarchy is a static, compile-time operation, Agrawal et al. also discuss the possibility of allowing incremental updates to the hierarchy. This aspect of the encoding is explored in Section 3.

Fall [Fal96] employs so-called *sparse terms* for the dynamic encoding of hierarchies from the top-down. Fall claims that the storage requirements for sparse terms is significantly less than that for bit vectors and intervals, although this is not apparent from the simple encoding in Figure 2. Although optimal for certain types of hierarchies, sparse terms are of the same order of storage as the other encodings on general binary trees. In fact, in the real-world example of 1,815 nodes from a chess-learning program, sparse terms required 442 bits per code, as compared to 590 for the compact encoding of Ait-Kaci et al. The measure of bits per code for sparse terms is not well explained, and appears to be an underestimate for the space required to keep track of sparse terms in a dynamic environment.

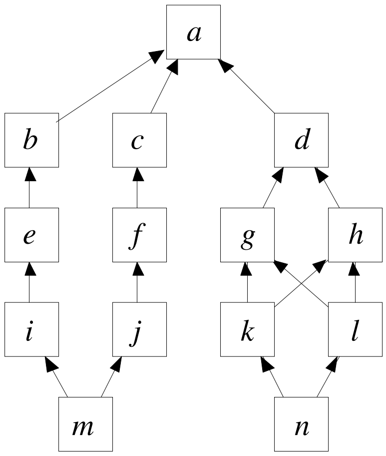


Figure 3: Example hierarchy.

# 3 Methods

The adaptations of the compact encoding method of Ait-Kaci et al. [AKBLN89] and the range compression scheme of Agrawal et al. [ABJ89] to persistent, incremental applications involves the handling of dynamic encoding starting from the top class in the hierarchy and encoding each new leaf class as it is added as a subclass of existing classes. That is, hierarchies will be encoded from the top down, with no new classes added in the middle of an existing inheritance relationship, and no new edges added between existing nodes. Since the encodings are performed at run-time, the amount of overhead being stored to assist in the encoding must be minimized. Further, the computation of the lattice operations must remain efficient. The methods are presented below, with the hierarchy illustrated in Figure 3 used for demonstration.

## 3.1 Incremental Top-down Encoding

Incremental top-down encoding arose from the application of the top-down approach of Caseau [Cas93] to the bottom-up encoding methods of Ait-Kaci et al. [AKBLN89]. It differs from the bottom-up approach in that it enables codes to be generated for each new class as it is added to the hierarchy, ridding the system of the overhead of code recomputation and assignment. In addition, the code lengths are made variable, so that those classes that only require a short code do not store a long string of zeros, thus saving additional storage. The overall goal of the encoding is to assign a binary code to each class such that the code of a class contains all of the bits in the code of any ancestor, with no class containing all of the bits of a class that is not an ancestor, which would indicate a false subsumption relationship.

The encoding proceeds as follows. The root node is assigned a code of 0, and each new class is assigned a code calculated using the functions in Figure 4, where it is assumed that the global variable $p$ is initially zero. The function *Encode* determines the new code for class $n$ by computing the binary OR of the codes of $n$'s parents. Each new code must be unique and not conflict with any other code already present in the hierarchy; that is, new codes should not imply any subsumption relationships not actually present. *ResolveConflicts* ensures this by comparing the code assigned to a new class to the codes attributed to the members in the incomparable set of the class. The incomparable set is made up of the classes that are neither subclasses nor superclasses of the given class. It is sufficient to examine the classes descended from the parents of the given node, since siblings of the parents and other ancestors are always assigned a distinct bit position by *ResolveConflicts*, making the codes of classes descended from them distinct from the new code. If a conflict is discovered during *ResolveConflicts*, *Increment* ensures that the codes involved are made unique by placing a bit at the beginning of each code. When a class with children is incremented, its descendants must also receive this bit to maintain the containment of the parent code in the descendant code. *Propagate* handles this procedure.

In *ResolveConflicts*, if the new code matches the code of another class, the two classes must be siblings, and both codes must be incremented to make them distinct for subsumption comparisons. If the code of one class is contained within that of another, and the two classes are not related, the code of the one must be incremented to contain a bit not found in the other. Finally, since by the assumption that classes are added only as leaf classes, class $x$ has no descendants, increments to the code for class $x$ need not be propagated to descendants.

The final encoding for the example hierarchy is given on the left side of Figure 5. After assigning code 0 to node $a$, *Increment* of the parent code is used to assign the codes to classes $b$ through $j$. The code for classes $k$ and $l$ result from the binary OR of the codes of their common parents but, since this would lead to the same code for the two classes, *ResolveConflicts* increments both of the codes ($k$ falls within the incomparable set as a descendant of the parent of $l$), and calls *Propagate* to to add the new bit to the descendants of class $k$. The codes for classes $m$ and $n$ are simply the binary OR of the codes for their respective parents.

With this method, the lattice operations of GLB and LUB become binary OR and AND operations, respectively. For example, for the GLB of $i$ and $j$, take the OR of 10001001 and 100010010, giving 110011011, which is precisely the code of $m$, the GLB of $i$ and $j$. It should be noted that the operations do not necessarily return the code of a class. Taking the GLB of $g$ and $h$ gives 1100100, which is the code of none of the classes. The decoding function

$$\gamma^{-1}(c) = \lceil \{x \mid c \sqsubseteq \gamma(x)\} \rceil$$

which extracts the top classes subsumed by the code gives the set $\{k, l\}$ as desired, which is the set of maximal common lower bounds of $g$ and $h$. Similarly, the GLB of $e$ and $f$ gives 11011, which results in the class $m$ after application of the decoding function.

Incremental top-down encoding has a best case space bound of $O(\log n)$ bits per node, and a worst case of $O(n)$, where $n$ is the number of nodes in the hierarchy. In the best case, the hierarchy is a complete lattice, and a node exists for each possible binary code. In the worst case, the hierarchy is a chain or single branch, or has only one level.

The best case time bound to encode a new node is constant time, where the code of a single parent is incremented. In the worst case, the time is $O(n)$

$Encode(x : class)$**::**
  **let** $\{x_1, ..., x_n\} = Parents(x)$ **in**
    **if** $n > 1$ **then**
      $\gamma(x) \leftarrow \bigvee_{i=1}^{n} \gamma(x_i)$
      $ResolveConflicts(x)$
    **else** $\gamma(x) \leftarrow Increment(x_n)$

$Propagate(x : class)$**::**
  **for each** $y \in Children(x)$ **do**
    $\gamma(y) \leftarrow \gamma(y) \vee \gamma(x)$
    $Propagate(y)$

$ResolveConflicts(x : class)$**::**
  **for each** $y \in IncSet(x)$ **do**
    **if** $\gamma(x) = \gamma(y)$ **then**
      $\gamma(x) \leftarrow Increment(x)$
      $\gamma(y) \leftarrow Increment(y)$
      $Propagate(y)$
    **else if** $\gamma(x) \wedge \gamma(y) = \gamma(x)$ **then**
      $\gamma(x) \leftarrow Increment(x)$
    **else if** $\gamma(x) \wedge \gamma(y) = \gamma(y)$ **then**
      $\gamma(y) \leftarrow Increment(y)$
      $Propagate(y)$

$Increment(x : class) : binary$**::**
  $p \leftarrow p + 1$
  **return** $2^{p-1} \vee \gamma(x)$

Figure 4: Incremental top-down encoding algorithm.

to compare the new code with every other class, in the case where the new node has two parents, and every other class is a sibling.

## 3.2 Modulated Top-down Encoding

To further compress the encoding for each class, the notion of grouping (a simplification of modulation [AKBLN89]) is used with incremental top-down encoding to allow classes to be divided into groups based on inheritance from classes at the top levels of the hierarchy. Each group is assigned a distinct code, separating it from other groups, and classes in the group are assigned codes as in top-down encoding. This enables the sharing of bit positions in class codes in different groups, while keeping the distinction of the group code.

Multiple inheritance creates difficulties with this scheme as an added class may have parents in two or more distinct groups. This requires a group combining operation. First, the code for the new group is created by taking the binary OR of the codes of the groups involved, and all members of the groups become members of the new group. Since the group code no longer

| **Class Name** | **Top-down Encoding** | **Modulated** Group | Code |
|---|---|---|---|
| $a$ | 0 | 0 | 0 |
| $b$ | 1 | 11 | 1000 |
| $c$ | 10 | 11 | 100 |
| $d$ | 100 | 100 | 0 |
| $e$ | 1001 | 11 | 1001 |
| $f$ | 10010 | 11 | 101 |
| $g$ | 100100 | 100 | 1 |
| $h$ | 1000100 | 100 | 10 |
| $i$ | 10001001 | 11 | 1011 |
| $j$ | 100010010 | 11 | 111 |
| $k$ | 10001100100 | 100 | 1011 |
| $l$ | 1001100100 | 100 | 111 |
| $m$ | 110011011 | 11 | 1111 |
| $n$ | 11001100100 | 100 | 1111 |

Figure 5: Top-down and modulated encoding results.

differentiates classes that share bit positions, the codes of the classes are adjusted with a distinct bit for each member in each group incorporated in the new group.

Consider the example hierarchy in Figure 3. Class $a$ is placed in the first group with code 0. Classes $b$, $c$, $d$ are placed in separate groups with codes 1, 10, and 100, respectively. Classes $b$, $e$, and $i$ end up with codes 0, 1, and 10 in group 1, while classes $c$, $f$, and $j$ end up with the same codes on group 10. The addition of class $m$ results in the combination of the two groups, leading to group 11, and the codes in group 10 incremented with a bit in position three, and those in group 1 with a bit in position four. The resulting codes for the entire hierarchy on the right side of Figure 5.

Adding further levels of grouping (that is, grouping within a group) can further compact the overall code size. Further reuse of bit positions could result, with the added expense of further considerations given to group combining with multiple inheritance. In large hierarchies, the benefits would be great. Unfortunately, the depth of grouping has its limitations. The number of classes in a group determines the overall efficiency; too many classes, too little code reuse; too few classes, too many groups and a large group code. Also, with too much grouping multiple inheritance would lead to many group combinations, and thus too many increases in the code lengths of the classes in the groups. Experimental results indicate that the grouping should only be performed at the second or third level.

The efficiency of the lattice operations suffers slightly with modulated encoding, as an additional comparison must be performed on the group codes to determine if the classes are in the same group. If the classes are in the same group, then the lattice operations are the same as with non-modulated encoding. Two classes in different groups where the group codes are in a subsumption relationship must also be in a subsumption relationship. Finally, the GLB and LUB of modulated codes is the binary OR and AND respectively of both the group and class codes. The decoding function does become a bit more complex as well, and involves determining the maximal elements whose codes are subsumed by the code, using the subsumption test of modulated codes.

As an example, consider the LUB of classes $i$ and $k$. The binary AND operation gives the group code 0 and class code 1011. The only element in the hierarchy subsumed by group code 0 and class code 1011 is class $a$, with group code 0 and class code 0. Examination of the hierarchy reveals that class $a$ is the LUB of $i$ and $k$.

## 3.3 Top-down Range Compression

Agrawal et al. [ABJ89] describe a modification to their range compression scheme which permits incremental updates to the hierarchy after an encoding has been performed. The idea is to leave gaps between the numbers used for nodes so as to permit the assignment of numbers in the gap to new nodes. A slight variation of this approach is described in this section which permits the hierarchy to be encoded in a completely incremental fashion. Index numbers are assigned to each node beginning with an arbitrarily high number for the root node. Subsequent numbers for each descendant are assigned based on gaps remaining in the numbers currently used. As long as the root index number is sufficiently high, the numbers available will not be exhausted, even with relatively large hierarchies.

In the Agrawal et al. encoding, each node is assigned a postorder number based on a spanning tree. An interval is recorded for each node representing the range of postorder numbers in its descendant list, including its own. In the case of multiple inheritance, multiple intervals must be recorded to represent disjoint intervals of descendants to include those connected by arcs not in the postorder traversal of the spanning tree (see Figure 2). Subsumption testing can be performed by determining if the postorder number of the descendant is covered by the interval(s) of the ancestor. The GLB for two classes is the intersection of their intervals, and LUB represents the union of the intervals. For example, in Figure 2, the postorder number of the descendant *TA*, 2, is covered by the interval of the ancestor *Student*, [1,3] and [4,5]. Also, the GLB of *Student*, [1,3] and [4,5], and *Employee*, [2,2] and [4,7], is the intersection [2,2] and [4,5], which covers the postorder numbers of *TA*, 2, and *RA*, 5.

In the modified algorithm, postorder numbers are called index numbers. In order to simplify the search for index numbers for nodes with multiple inheritance, a new node added to the hierarchy with multiple parents has one of the parents designated as the primary parent. The other parents are ignored in the determination of index number. The goal is to assign index numbers so that the increasing sequence of their values represent a postorder traversal of the tree resulting from the removal of non-primary parent links.

In the experiments, the root node is assigned the value $2^{16} - 1$. This value turns out to be practical with the use of 16-bit integers and with the sizes of the hierarchies involved. For each subsequent node, the index number is assigned so as to maximize the remaining available index numbers for descendants of each existing node. The interval assigned has the index number as both its low and high values. The intervals of the node's ancestors must be modified to include this new value if they do not already cover it.

The procedure is informally described using the example hierarchy and a largest index number of 64. The top node, $a$, is assigned the index number 64, and the interval [64,64]. The index numbers for subsequent nodes are determined in one of two ways. If the new node has siblings with the same primary parent, then the middle value of the largest gap uncovered by the intervals of the parent and siblings is used as the new index number. By this method, nodes $b$, $c$, and $d$ are assigned index numbers 32, 48, and 16, respectively.

If the new node has no siblings with the same primary parent, then the closest primary ancestor with siblings with the same primary parent is found. The new index number is taken from the middle of the gap between the index number of the primary parent of the new node and the sibling with the largest index number less than this value. If there is no sibling with a index number less than this value, than the gap used consists of the smallest possible index number and that of the primary parent of the new node. This method assigns index numbers 24, 40, and 8 to nodes $e$, $f$, and $g$. These numbers must be included in the intervals of the ancestors, thus setting the intervals of nodes $a$, $b$, $c$, and $d$ to be [8,64], [24,32], [40,48], and [8,16], respectively.

Nodes $h$, $i$, and $j$ get assigned index numbers 12, 20, and 36, respectively using these methods. As well, the intervals get adjusted to reflect those in the first column of Figure 6.

Adding a node with multiple parents involves the same procedure for the primary parent, with the additional requirement of adding the new interval for the node to the other parent(s) and their ancestors, if necessary. For example, the addition of node $k$ with parent $g$ designated its primary parent gives it index number 4, and requires the extension of the intervals of the ancestors $a$, $d$, and $g$ to include 4. For the secondary parent link to node $h$, the interval [4,4] is added to $h$ as a second interval. This need not be added to the ancestors of $h$ since they were already made to include the interval. The case is similar for class $l$ with primary parent $h$, except that some of the ancestors already include the new index number 10 in their intervals. This leads to the second encoding in Figure 6.

The addition of class $m$ with primary parent $j$ gives it index number 34, which adds second intervals to $i$ and its ancestors $b$ and $e$ and adds 34 to the intervals of $c$, $f$, and $j$. The final change is the addition of node $n$ with primary parent $l$, which is assigned index number 9. This adds the second interval to class $k$ and extends the intervals of $g$, $h$, and $l$.

Top-down range compression in the best case uses constant space for each node, and a worst case space bound of of $O(n)$ for each node, where $n$ is the number of nodes in the hierarchy. In the best case, no multiple inheritance occurs, and the hierarchy is a tree. The worst case involves a bipartite graph, with one interval associated with each of the $m$ bottom nodes and one top node, and $m + 1$ intervals associated with the remaining top nodes.

The best case time bound to encode a new node is constant time, where the node has a single sibling. In the worst case, the time is $O(n^2)$, where the hierarchy has one level and all nodes must be checked to determine the new index number.

| | | | |
|---|---|---|---|
| $a$ | [ 8,64] | [ 4,64] | [ 4,64] |
| $b$ | [20,32] | [20,32] | [20,32],[34,34] |
| $c$ | [36,48] | [36,48] | [34,48] |
| $d$ | [ 8,16] | [ 4,16] | [ 4,16] |
| $e$ | [20,24] | [20,24] | [20,24],[34,34] |
| $f$ | [36,40] | [36,40] | [34,40] |
| $g$ | [ 8, 8] | [ 4, 8],[10,10] | [ 4, 8],[ 9,10] |
| $h$ | [12,12] | [10,12],[ 4, 4] | [ 9,12],[ 4, 4] |
| $i$ | [20,20] | [20,20] | [20,20],[34,34] |
| $j$ | [36,36] | [36,36] | [34,36] |
| $k$ | | [ 4, 4] | [ 4, 4],[ 9, 9] |
| $l$ | | [10,10] | [ 9,10] |
| $m$ | | | [34,34] |
| $n$ | | | [ 9, 9] |

Figure 6: Incremental range compression results.

# 4 Experimental Results

To compare the encoding methods in the previous section, several hundred hierarchies were generated. For each hierarchy, there was a single top-level node, and the number of second level nodes (between four and ten) was supplied. The resulting hierarchies were pseudo-randomly generated based on a multiple inheritance probability factor of 10% and with an average depth of seven levels. This produced 380 hierarchies ranging in size from seven classes to 458 classes. Figure 7 illustrates the results of these experiments by comparing the number of classes in a hierarchy with the total number of bytes required for storing the encoding.

As expected, non-modulated top-down encoding performs poorly for most hierarchies, due to the rapid increase in code size as the hierarchies become wider. Modulated encoding reduces the code size significantly, but storage is still significantly larger than with range compression. Range compression produces a roughly linear relationship between the number of classes and storage size, since each class receives a discrete number of bytes per interval and the number of intervals per class remains small. Few classes require multiple intervals due the the relatively small multiple inheritance factor used in the experiments, which is typical of most real-world applications [Cas93].

A significant result for this work is how well sub-modulated encoding performed. Sub-modulated encoding involves a second level of grouping; that is, groups are created at the third level of the hierarchy. Over 80% of the hierarchies encoded achieved the lowest storage with sub-modulation, which is significant considering the linear nature of range compression. On hierarchies with more than 300 classes, range compression achieved a more compact encoding, though the savings are at the added cost of more complex lattice operations.

To determine if these results were consistent with a real-world hierarchy, a 300-object LAURE class lattice [Cas91] was obtained and encoded in a top-down manner. Non-modulated top-down encoding produced a code of 5,279 bytes, with the largest code being 281 bits. A top level of modulation reduced the size to 5,247, and 279 bits for the largest code; a small change

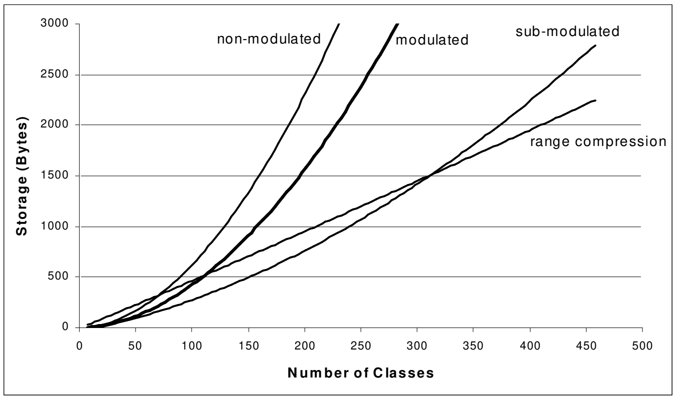


Figure 7: Results on randomly generated hierarchies.

due to the fact that the top level has only four classes and multiple inheritance ends up combining all four of the resulting groups. A second level of modulation results in a size of 4,403, with 252 bits for the largest code, and only two groups. The most benefit is obtained with a third level of modulation where the resulting size is 1,452 bytes, with a largest code size of only 120 bits, and 30 groups. This result is comparable to that of range compression, where the encoding requires 1,336 bytes.

One further note on the two encoding methods is necessary. For continued growth of a large hierarchy, the choice of index number for the root node in range compression may become insufficient; that is, there may be a point in time where there is no remaining numbers to choose for a new class. In this situation, the current values will have to be adjusted - an expensive operation. With top-down incremental encoding, the number of bits being used for each class is variable, and thus the codes for new classes can be expanded without affecting the existing codes.

## 5 Summary

Persistent applications involving multiple inheritance hierarchies require the ability to efficiently compute lattice operations via some encoding scheme. A second requirement is the ability to allow incremental update to the hierarchy without the entire recomputation of the encoding and the reassignment of codes to classes. Previous work on encoding has focused on compile-time encodings, requiring the recalculation of codes once new classes are added. As well, the static nature of these methods permits significant overhead to be maintained during the encoding process.

This paper has examined several top-down, incremental encoding methodologies. Comparisons made illustrate that there is a trade-off in the final choice for a particular application. For small hierarchies (less than 300 nodes), a top-down incremental encoding with second level grouping (modulation) appears to perform best. For larger hierarchies, top-down range compression achieves a more compact encoding, with the added expense of slightly more complex lattice operations. Unfortunately range compression, as implemented, is unable to cope with very large growth of a hierarchy.

One benefit of the encodings is the ability to represent the union or disjunction of classes as a single code or interval. For example, for top-down incremental encoding, the GLB of classes $g$ and $h$ of the example hierarchy produced the code 1100100, which represents the code of no single class. Using the decoding function this code was calculated to represent the disjunction of $k$ and $l$. In some applications, the storage of the code without decoding is sufficient. For example, in the database constraint reasoning system developed in [vB96], the code is sufficient to determine which constraints apply to a particular object; thus decoding is only performed when required in the output.

Work is on-going to determine if relaxing the need for the LUB operation can shorten the codes required in incremental encoding. With the close relationship to the work of Caseau [Cas93], perhaps the resulting code size will be significantly smaller. Further, work is continuing on implementing the encodings into the constraint reasoning system mentioned above.

## Acknowledgements

The authors are grateful for the comments of the editor and anonymous reviewers which helped clarify the description of the implementations and the overall text. Also, we wish to acknowledge the support of the Natural Science and Engineering Research Council of Canada, and the St. Francis Xavier University Council for Research.

## References

[ABJ89] R. Agrawal, A. Borgida, and J. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 253–262, June 1989.

[AKBLN89] H. Ait-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, January 1989.

[Cas91] Y. Caseau. An object-oriented deductive language. *Annals of Mathematics and Artificial Intelligence*, 3(2), March 1991.

[Cas93] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In *Proceedings of the International Conference on Object-Oriented Systems, Languages, and Applications*, pages 271–287, October 1993.

[Fal96] A. Fall. Sparse term encoding for dynamic taxonomies. In *Proceedings of the Fourth International Conference on Conceptual Structures*, Sydney, Australia, 1996.

[GMR94] D. Ganguly, C. Mohan, and S. Ranka. A space-and-time efficient coding algorithm for lattice computations. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):819–829, October 1994.

[vB96] M. F. van Bommel. *Path Constraints for Graph-Based Data Models.* PhD thesis, Department of Computer Science, University of Waterlo, 1996.