Incremental Encoding of Multiple Inheritance Hierarchies Supporting Lattice Operations

M.F. van Bommel and T.J. Beck
Department of Mathematics, Statistics, and Computer Science
St. Francis Xavier University
Antigonish, Nova Scotia B2G 2W5 CANADA
Phone: (902)867-3857 Fax: (902)867-2448
email: {mvanbomm,x93gtd}@stfx.ca

Abstract

Incremental updates to multiple inheritance hierarchies are becoming more prevalent with the increasing number of persistent applications supporting complex objects. Efficient computation of lattice operations such as greatest lower bound (GLB), least upper bound (LUB), and subsumption subsequently is becoming more and more important. General techniques for the compact encoding of a hierarchy are presented that support the operations, and are flexible enough to allow incremental updates to the hierarchy. One such method is to plunge the given ordering into a boolean lattice of binary words, leading to an almost constant time complexity of the lattice operations. The method is based on an inverted version of the encoding of Ait-Kaci et al. to allow incremental update. Simple grouping is used to reduce the code space while keeping the lattice operations efficient. Comparisons are made to an incremental version of the range compression scheme of Agrawal et al., where each class is assigned a interval, and relationships are based on containment in the interval. The result is two encoding methods which have their relative merits.

Keywords

Multiple inheritance hierarchy, lattice operations, incremental update, top-down encoding.

Category: Long paper.

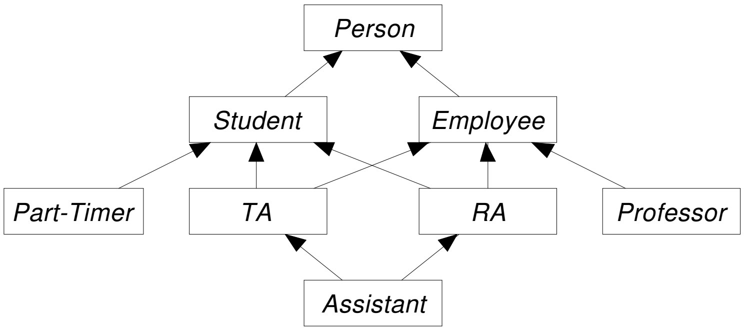

Figure 1: Partial university hierarchy.

1 Introduction

In the evolving world of computer science, applications for hierarchical structures are arising in numerous research areas. Artificial Intelligence presents many such applications. Neural network models, semantic nets and state spaces all require such a structure. Object oriented programming languages such as C++ and Java, which are based on the relationship between classes and instances of these classes, called objects, are prime candidates for an organization involving hierarchies. Complex-object databases, scheduling packages, and computer aided design software such as UML (Unified Modeling Language) and OMT (object modeling technique) all employ hierarchical structures. All of these areas have one problem in common. They all have a need to represent relationships between classes of objects in an efficient manner, and to be able to compute these relationships efficiently.

In general, 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 a possibility of the result representing a disjunction or conjunction of classes, respectively.

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**	**Caseau**	**Agrawal**
Person	11111	00000	[1,8]
Student	11011	00001	[1,3], [4,5]
Employee	11101	00010	[4,7], [2,2]
Part-Timer	00010	00101	[1-1]
TA	01001	01011	[2,2], [4,4
RA	10001	10011	[4,5]
Professor	00100	00110	[6,6]
Assistant	00001	11011	[4,4]

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. Recomputing the encoding is not only too time consuming, but also requires reassigning new codes to objects in persistent applications. The ideal encoding allows for additions to the hierarchy and few changes to existing codes.

Two earlier methods are reviewed in Section 2, along with a third which fails to support GLB and LUB operations. Variations of the two 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 prohibits 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.

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 that 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 hierarhy. 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.

3 Methods

The adaptation of the compact encoding method of Ait-Kaci et al. [AKBLN89] and Agrawal et al. [ABJ89] to a persistant, incremental application involves the handling of dynamic encoding starting from the top class in the hierarchy and adding 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. 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

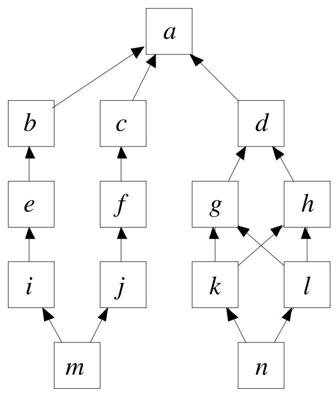

Figure 3: Example hierarchy.

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 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. 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. *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. If a conflict arises, *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. *Propogate* handles this procedure.

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 to j. The code for classes k and l result from the binary OR of the codes for their parents but, since this would lead to the same code for the two classes, the call to *ResolveConflicts* increments the code for class l as a member of the incomparable set, and calls *Propagate* to increment the code for 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

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

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

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

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

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

Figure 4: Incremental top-down encoding algorithm.

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.

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 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.

Class Name	Top-down Encoding	Modulated Group	Modulated 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.

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 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.

The root node is assigned the value $2^{16} - 1$ in our experiments. This value turned 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 functions in Figure 6 perform the encoding. *FindPlace* calculates the correct index number by comparing the differences between numbers among the primary parent of the node and all of the node's siblings. A lower limit for the new index number is also calculated so that the ranges do not overlap. *FindLowerLimit* determines the index number of the next sibling of the first ancestor with siblings. The two existing nodes with the largest difference in index numbers become the bounds for the new index number. If the largest difference occurs between the lowest value and the lower limit, these become the bounds. The value half way between the bounds maximizes the remaining available index numbers, and thus is used as the new index number for the new node. The range assigned to the node uses this number as both its lower and upper bounds. The lower bounds of the nodes ancestors are changed to this new value if they do not already contain it in their range. This adjustment is performed by *AdjustRanges*.

In a multiple inheritance situation the index number of the new node is determined solely based on one of its parents (its primary parent). The other parent(s) are ignored initially. After the node has had a range assigned to it, the remaining parent(s) and their ancestors must have the range of the new node added to their set of ranges. Ancestors who already encompass this range do not require the additional range.

Consider the example hierarchy. Class a is assigned the index number 65535. Classes b, c, and d are then assigned numbers 32767, 49151, and 16383, respectively, by maximizing the gaps between values. Classes e, f, g, h, i, and j, each of which requires an index number less than that of its parent and greater than that of its parent's siblings, are assigned 24575, 40959, 8191, 12287, 20479, and 36863, respectively. This leads to the encoding on the left side of Figure 6, where the range assigned to each class consists of the lowest index assigned to its descendants and its own index. Adding class k with index number 4095 requires adding the second range to h due to the multiple inheritance, as well as the extension of the ranges of the ancestors a, d, and g to include 4095. The case is similar for the addition of class l, except that some of the ancestors already include the new index number 10239 in their ranges. This leads to the second encoding in Figure 6. The addition of class m with index number 34815 adds second ranges to i and its ancestors b and e and adds 34815 to the ranges

a	[8191,65535]	[4095,65535]	[4095,65535]
b	[20479,32767]	[20479,32767]	[20479,32767],[34815,34815]
c	[36863,49151]	[36863,49151]	[34815,49151]
d	[8191,16383]	[4095,16383]	[4095,16383]
e	[20479,24575]	[20479,24575]	[20479,24575],[34815,34815]
f	[36863,40949]	[36863,40949]	[34815,40949]
g	[8191, 8191]	[4095, 8191],[10239,10239]	[4095, 8191],[9215,10239]
h	[12287,12287]	[10239,12287],[4095, 4095]	[9215,12287],[4095, 4095]
i	[20479,20479]	[20479,20479]	[20479,20479],[34815,34815]
j	[36863,36863]	[36863,36863]	[34815,36863]
k		[4095, 4095]	[4095, 4095],[9215, 9215]
l		[10239,10239]	[9215,10239]
m			[34815,34815]
n			[9215, 9215]

Figure 6: Incremental range compression results.

of c, f, and j. The final change is the addition of node n with index number 9215. This adds the second range to class k and extends the ranges of g, h, and l. This is illustrated in the final encoding of Figure 6.

The lattice operations for range compression are not as simple as those for top-down encoding, but they remain efficient. Subsumption can be calculated simply by comparing the ranges of each class involved. For example, classes n and h are in a subsumption relationship, as illustrated by the fact that the index number of class n, 9215, is included in one of the ranges of class h, [9215,12287]. The GLB of two classes is calculated as the intersection of their ranges. For example, the GLB of classes i and j is calculated by intersecting the ranges of i, namely [20479,20479] and [34815,34815] with the range of j, namely [34815,36863], giving the range [34815,34815], which is precisely the range of class m, the GLB of i and j. As with top-down encoding, the resulting range may not represent a single class. The GLB of g and h gives the resulting ranges [4095,4095] and [9215,10239], which represents the classes k and l as the set of top classes subsumed by the ranges. LUB can be calculated similarly using the union of the ranges.

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

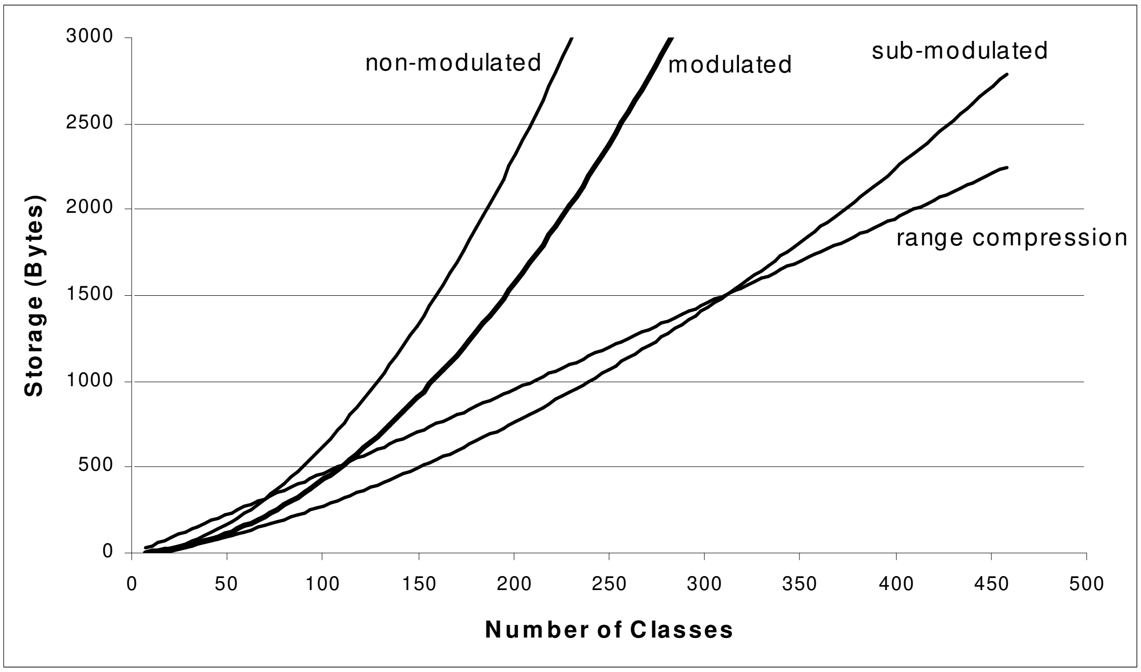

Figure 7: Results on randomly generated hierarchies.

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 drastically, 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 range and the number of ranges per class remains small.

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 using the sub-modulation method, 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 involved in the 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 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 works on encoding have 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 range. 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.

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.

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