Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices

Robert Godin, Hafedh Mili
Département de Mathématiques et d'Informatique
Université du Québec à Montréal
Case Postale 8888 (A)
Montréal, PQ H3C 3P8
CANADA

Abstract

Software reuse is one of the most advertised advantages of object-orientation. Inheritance, in all its forms, plays an important part in achieving greater reuse, at all stages of development. Class hierarchies start taking shape at the analysis level, where classes that share *application-significant* data and *application-meaningful external behavior* are grouped under more *general* classes. At the design level, such hierarchies are augmented with implementation classes, and possibly reorganized to take into account implementation factors such as performance or code reuse [Rumbaugh91a]. Getting the analysis-level hierarchy "right" is very important for the understandability and traceability of the models and the reusability of the resulting code [Rumbaugh91a]. In this paper, we propose a formal method that organizes a set of class interfaces into a lattice structure called *Galois Lattice* [Godin86a]. Such a lattice has several advantages including: 1) embodying protocol conformance, 2) supporting an incremental updating algorithm [Godin93a]), with applications for class hierarchy maintenance. We first present the basic method and illustrate its use through an example inspired from [Cook92a]. Next, we discuss extensions to the method to take into account richer class descriptions in general, and the specifics of OO analysis-level models. Finally, we discuss some of the research directions we are currently pursuing.

1. Introduction

Several tenets of object-orientation make OO software more reusable: 1) *information hiding*, which shields clients (objects, modules) from implementation changes in servers, 2) *genericity* and *overloading*, which parametrize functionality, by abstracting out some type-dependencies, and 3) *inheritance*, which provides a conceptual framework (*and a language construct*) for *reusing* software components. Inheritance, plays an important part in achieving greater reuse, at all stages of development. Class hierarchies start taking shape at the analysis level, where classes that share *application-significant* data and *application-meaningful external behavior* are grouped under more *general* classes. Identifying generalizations of classes at this level has several advantages including: 1) enhancing the understandability of the models by reducing the number of *independent* concepts that an analyst/user has to deal with, 2) providing a cross-check with data dictionaries to enforce consistency within the model, and 3) identifying opportunities for code reuse [Wirfs-Brock90a, Rumbaugh91a, Coad91a]. The latter is justified by the intuitive realization that similar requirements in terms of external behaviors-- an analysis-level product-- generally lead to similar implementations. In this paper, we propose a family of methods that organize classes within a lattice based on the specification of their interfaces.

Existing OO development methodologies (see e.g. [Rumbaugh91a, Wirfs-Brock90a, Coad91a]) prescribe that design builds on the basic class structure identified at the analysis level-- one aspect of the much vaunted seamless transition-- by adding high level (control) and low level (utility)

OOPSLA'93, pp. 394-410

application-independent classes. Methodologists recognize that in some cases, some restructuring of the basic application classes may be warranted to accommodate some implementation-level concerns such as performance and code reuse. They also suggest looking into other alternatives to inheritance (e.g. *delegation*) that achieve the same goals [Rumbaugh91a]. A number of researchers have observed that existing class hierarchies do not always "make sense" (see e.g. [Cox90a, Cook92a]). Cook thoroughly studied Smalltalk-80's collection classes and found a number of discrepancies between the protocols that classes implement and their place in the code class hierarchy. He explains the discrepancy by the fact that the (code) class hierarchy presents the "implementers view" of the class library, while the protocol hierarchy presents the "clients view" [1][Cook92a]. In [Cox90a], Brad Cox studied a commercial class library and dwelled on the extent to which the place of some classes in the hierarchy did not make sense:

> "... Semaphores are a kind of Queue only from the arcane viewpoint of their author. This hierarchy resulted from a *speed optimization* of no interest to consumers, who should view as scheduling primitives with only wait and signal methods "[Cox90a].

In both cases, a reuser/user of the class library is a *client* who approaches the class library more from a requirements (analysis-level) point of view than from an implementation point of view.

While design and implementation efficiency may, in some cases, supersedes semantic/conceptual clarity, we believe that "unnatural" class (code) hierarchies often result from poor design choices, as in using inheritance when delegation is more appropriate [Rumbaugh91a], or in poor analysis-level classification, i.e., all implementation characteristics equal (e.g. space requirements, time performance), a better class hierarchy could have been devised that achieves the same or a higher level of code reuse, with less cancellations, redefinitions, etc. Cook showed that Smalltalk's Collection class hierarchy can be improved-- semantically-- with no additional implementation costs [Cook92a].

Despite the importance of the analysis-level hierarchical organization of classes on the OO development lifecycle, and the long-established research tradition in classification in artificial intelligence, there have been notably few efforts to provide automated or semi-automated tools for building or maintaining class hierarchies. One such effort is part of the Demeter System, and involves the automatic discovery of classes from example objects, and the hierarchical organization of those classes [Bergstein91a]. They propose a two-step *learning algorithm*, where the second step, called the *minimization step*, consists of factoring out the common parts in a way that "optimizes" some structural properties of the *class dictionary graph* [Bergstein91a]. The concept of Galois lattice can be seen as a simple and elegant framework for dealing with the minimization step. Further, their method takes into account only data attributes-- no behaviors. Cook used automatically extracted interface descriptions of the Smalltalk80 collection classes to build an interface hierarchy based on interface conformance [Cook92a]. Other work addressed issues of class hierarchy maintenance, for example when new classes are added and new generalizations emerge (see e.g. [Pedersen89a, Gibbs90a, Bergstein91a], and to some extent [Ossher92a]), or when new methods are added to existing classes [Li89a, Ossher92a]. Such reorganization/maintenance methods incur local changes that plug unto existing hierarchies, although the underlying restructuring principles may-- modulo some changes-- be used to build hierarchies from scratch. The family of conceptual structures presented in this paper subsumes the ones used by Cook [Cook92a] and has interesting computational properties.

In section 2, we describe the basic method used to generate *Galois lattices* from class interfaces. We illustrate the method using a subset of the

1. This is possible in Smalltalk because it is untyped and because a class can "cancel" methods inherited from its superclasses (simply by redefining them to raise exceptions when they are invoked).

collection classes used by Cook [Cook92a]. In section 3, we describe extensions to the basic method that take into account richer class descriptions and application domain knowledge, thereby refining the classification. We conclude in section by summarizing the findings of the paper and discussing some research directions we are currently exploring.

2. Building a Galois Lattice of Class Protocols

The use of Galois[2] lattices as a model of conceptual hierarchies has been first introduced by Wille [Wille82a]. He used the Galois lattice of a binary relation between a set of instances and a set of features, as a basis for what he called *formal concept analysis* [Wille82a]. The Galois lattice of a binary relation between instances and features may be considered as an (extensional) class hierarchy where each node of the lattice represents a class in terms of: 1) the set of its instances, and 2) the set of shared features between its instances. Such lattices have been used for a variety of applications, including conceptual distance measurements [Ganter86a], and document classification and browsing [Godin86a, Godin93a]. Building such lattices incrementally may be likened to a process of learning or *concept formation* [Gennari90a]. We developed one such algorithm that incrementally generates/augments the Galois lattice and its Hasse diagram with an average case complexity of O(n), where n is the number of instances [Godin91a]. For reference, we illustrate the basic method and some flavors thereof on an example based on [Cook92a]. We describe the *full* Galois lattice in section 2.1. Two "refinements" are discussed in sections 2.2 and 2.3 to the extent that they relate to protocol hierarchies and class design.

2.1. The Galois Lattice of a Binary Relation

Consider two finite sets E (set of *instances*) and E' (set of *features*), and a binary relation R between the two. Among all the subsets X of E, only a few satisfy the property:

$$X = \bigcap_{y \in \bigcap_{x \in X} R(x)} R^{-1}(y)$$

where R(x) ($R^{-1}(y)$) is the set of images (antecedents) of x (y) by R. The same is true for subsets of E', where we interchange R and R^{-1}, and there is a one-to-one correspondence between the two. Roughly speaking, the Galois lattice of R is defined by the subset relation between those subsets of E (E') that satisfy the above property. Consider the binary relation represented by the boolean matrix in Figure 1. In this case, the set E (columns) correspond to classes and the set E' corresponds to the operations/methods supported by any one of those classes. The set of classes is a subset of the Smalltalk-80 ™ Collection classes that were studied by Cook in [Cook92a]. A sufficient subset of methods was chosen in order to relate the resulting structure to that given in [Cook92a]. For a given class, the *protocol* consists of the names of the operations (or *message selectors*) supported by the class[3]. Figure 2 shows the corresponding Galois lattice. Only the uppercase letters for the class names appear in the Figure.

2. Galois is a French mathematician of the early 20th century, who died, presumably, in a duel for the love of a young lady.

TM. Smalltalk-80 is a trademark of ParcPlace.

3. Smalltalk-80's keyword notation for methods with parameters was not respected for clarity. For example, **atPut** should read **at:put:**, or, **at: anIndex put:anObject**.

	Collection	Set	Bag	Sequence-able Collection	Dictionary	Linked List	Array
isEmpty	1	1	1	1	1	1	1
size	1	1	1	1	1	1	1
includes	1	1	1	1	1	1	1
add	0	1	1	0	1	1	0
remove	0	1	1	0	0	1	0
minus	0	1	0	0	1	0	0
addWithOccurrences	0	0	1	0	0	0	0
at	0	0	0	1	1	1	1
atPut	0	0	0	0	1	0	1
atAllPut	0	0	0	0	0	0	1
first	0	0	0	1	0	1	1
last	0	0	0	1	0	1	1
addFirst	0	0	0	0	0	1	0
addLast	0	0	0	0	0	1	0
keys	0	0	0	0	1	0	0
values	0	0	0	0	1	0	0

Figure 1. Matrix representation of a binary relation R.

Each element of the lattice is a pair-- called *concept* by Wille [Wille82a]-- (X,X'), where $X \subseteq E$, $X' \subseteq E'$, and:

1) $X' = f(X)$, where $f(X) = \{x' \in E' \mid \forall\ x \in X, (x,x') \in R\} = \bigcap_{x \in X} R(x)$, and

2) $X = f'(X')$, where $f'(X') = \{x \in E \mid \forall\ x' \in X', (x,x') \in R\} = \bigcap_{x' \in X'} R^{-1}(x')$

We say in this case that (X,X') is *complete*. Only *maximally extended pairs* are kept in the lattice. A set $X \subseteq E$ is considered to not be maximally extended if there are other instances (elements of E), that are not in X, but that are nevertheless described by *all* the features common to *all* the elements of X; X must be extended to include such instances. Conversely, if X' does not contain a feature x' that is shared by *all* of the elements of X, then it has to be extended to include x'. The pair of functions (f,f') represents a *Galois connection* between the power set of E (2^E) and the power set of E' ($2^{E'}$). The Galois lattice G for the relation R is the set of complete pairs related by the partial order defined as follows:

Let $C_1 = (X_1,X'_1)$ and $C_2 = (X_2,X'_2)$, we have:

$$C_1 < C_2 \equiv X'_1 \subseteq X'_2$$

Notice that the functions f and f' are monotonically decreasing in the sense that:

$$X'_1 \subseteq X'_2 \longleftrightarrow X_2 \subseteq X_1$$

and thus:

$$C_1 < C2 \equiv X_2 \subseteq X_1$$

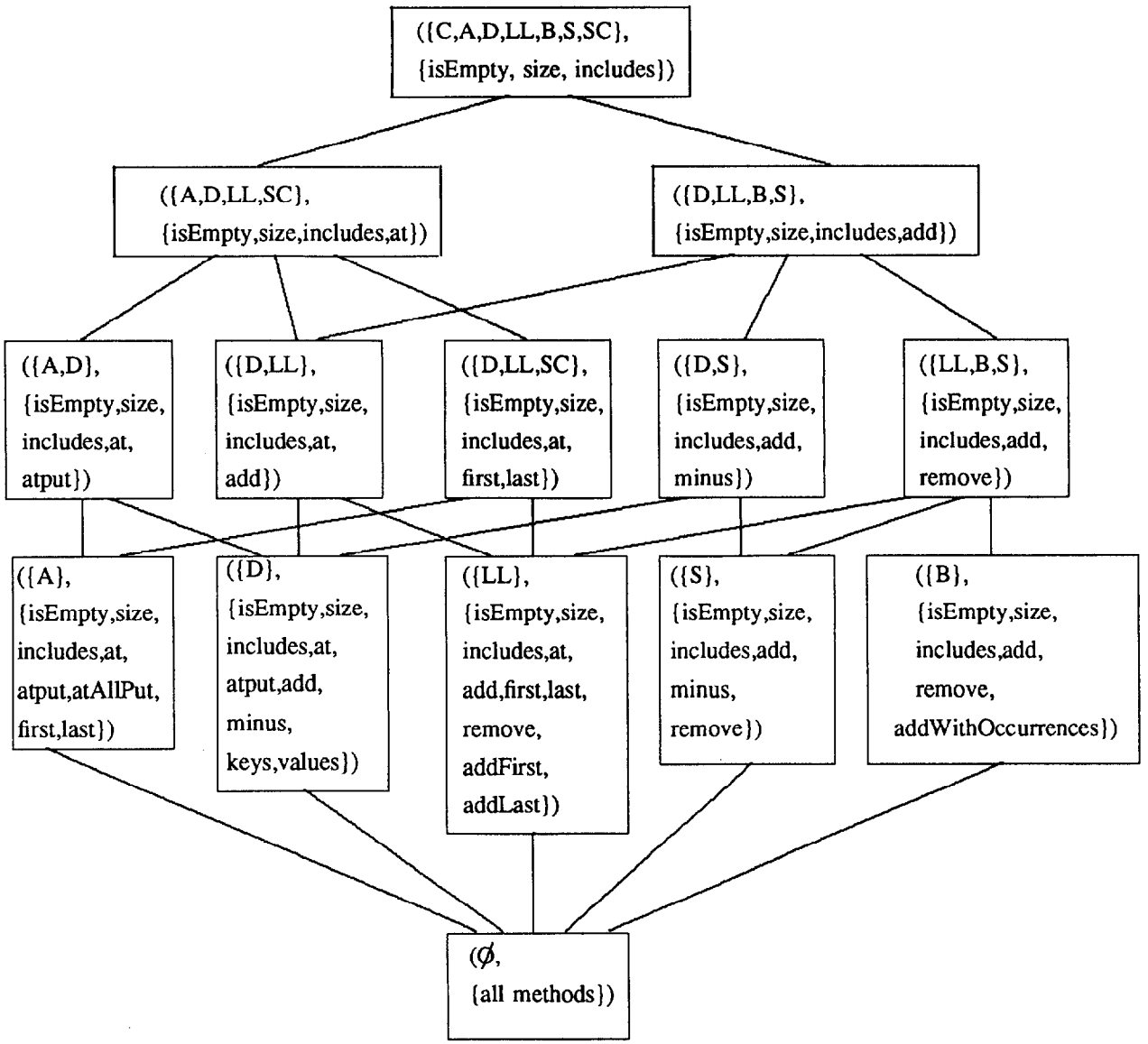


Figure 2. Galois lattice for the relation in Figure 1.

The *Hasse diagram* of the partial order "<" is generated as follows: we create an edge between C_1 and C_2, if $C_1 < C_2$, and if there is no other element C_3 in the lattice such that $C_1 < C_3 < C_2$. In other words, we create an edge between C_1 and C_2 if C_2 is the "smallest" pair that is bigger than C_1. When drawing a Hasse diagram, the edge direction is downwards. Given, C, a set of elements from G, inf(C) and sup(C) will denote respectively the greatest lower bound and lowest upper bound of the elements in C.

Each pair represents a set of classes with their common protocol and the Hasse diagram corresponds to the conformance relationship as defined by the inclusion of the sets of message selectors understood/supported by the classes [Cook92a]. Modulo some differences-- to be explained below--, the Galois lattice is closely related to the protocol hierarchy defined in [Cook92a]. That hierarchy was used as a tool for

analyzing and discovering inconsistencies in an *existing* class library. We suggest that such a hierarchy/lattice be generated *before* class design and implementation based on the operations that we wish the classes to support, and perhaps "try harder" to make the implementation hierarchy fit in the protocol hierarchy to avoid the kinds of problems discussed in [Cook92a].

Notice that while the number of subsets of E is exponential in the size of E, provided that the number of features per instance (i.e. cardinality of R(x)) is bounded-- which is usually the case in practical applications--, the worst case complexity of the structure is linearly bounded with respect to the number of instances (size of E) [Godin86a]. Further, we developed incremental algorithms for updating the structure, either by adding/removing instances, or by adding/removing features to existing instances. In our case, this would correspond to incorporating new classes in the protocol hierarchy, or adding/removing operations to existing classes. Empirical data presented in [Godin91a] showed that adding a new instance takes O(n) time, where n is the number of existing instances. Under the assumption of a fixed upper bound on the number of features per instance, this is also confirmed by a complexity analysis of the algorithm.

2.2. Inheritance Galois Lattice

There is much redundant information in a Galois lattice. For a pair C= (X, X'), X will be present in every ancestor of C and symmetrically, X' will appear in every descendant. The inherited (redundant) elements may therefore be eliminated without losing any information. For a pair C = (X, X'), let X'' be the set of elements in X that do not appear in any ancestor of X, and X''' the set of elements of X' that do not appear in any descendant of X'. The reader can check that, for a given complete pair (X,X'), X'' and X''' are defined as follows:

* $X'' = \{ x \in E \mid R(x) = X' \}$
* $X''' = \{ x' \in E' \mid R^{-1}(x') = X \}$

Figure 3 shows the lattice of Figure 2 where the nodes contain the pairs (X'',X''') instead of the pairs (X,X'). We call this lattice an *inheritance Galois lattice*. Notice that for a node C, the corresponding values of (X,X') can be recovered from the pairs (X'',X''') by taking the union of the X'' (respectively X''') sets for the descendants (respectively ancestors) of C, including C itself.

Depending on the application and on time versus space requirements, these structures may be considered as an alternative to the full Galois lattice representation. The representation used in [Cook92a] is based on the inheritance Galois lattice. For a given node, the inheritance Galois lattice shows *explicitly* only the message selectors that are not inherited from the parent nodes, and that are specific to the set of classes represented by that node. This may help clarify the concept represented by the node. For example, the node $(\Phi,\{\mathbf{at}\})$ can be interpreted as the set of collections whose elements can be accessed by position/index. We may call these **Indexed Collections**. Notice that a node with a pair **(Φ, X''') corresponds to the case where no class exists that implements** *exactly* the operations in X''' and the node's ancestors. In other words, there are no objects/instances in the application that will respond to only those messages. Accordingly, such a node can be considered to represent an *abstract class*. Figure 4 shows the named nodes, where we used the names in [Cook92a].

2.3. The Galois Knowledge Space (KS)

There remains an interesting difference between our lattice and that in Cook [Cook92a]: the node **IndexedExtensibleCollection**. Such a node has the peculiar property that both X'' and X''' are empty. It means that it represents an abstract class in the sense described above. Further, it does not add/introduce any operation that is not inherited from the ancestors. In some applications, such nodes are of no interest.

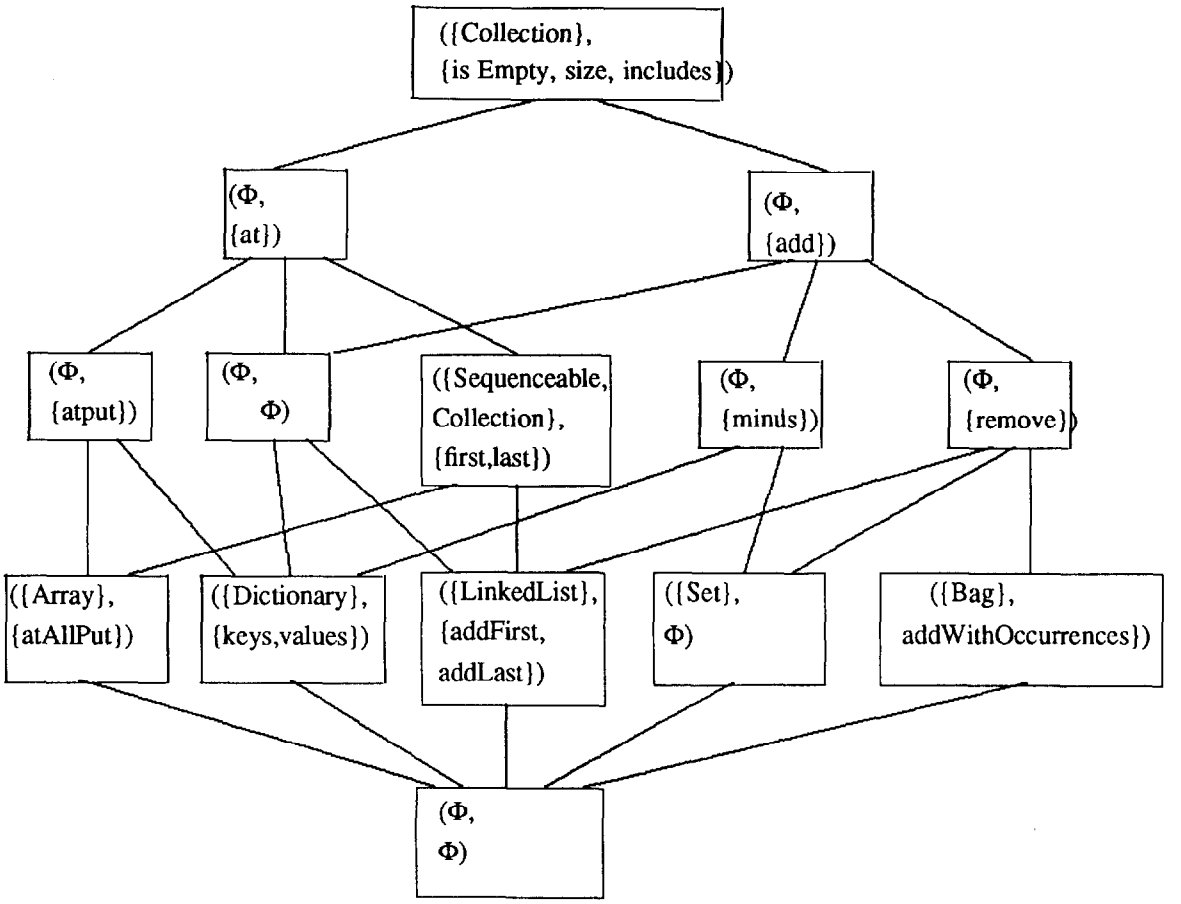


Figure 3. Inheritance Galois lattice.

Eliminating such nodes ("bypassing" them in the lattice) yields what we called a *knowledge space* (KS)[4], and has been used in the more general context of conceptual clustering of conceptual graphs [Mineau90a]. For the sake of illustration, the resulting hierarchy is shown in Figure 5, which coincides with that in [Cook92a].

Whether such nodes (i.e. (Φ,Φ)) should be deleted from the inheritance Galois lattice is actually debatable. First, as a model of the application domain, the full inheritance Galois lattice (Figure 4)-- or for that matter the full Galois lattice (Figure 2) with the appropriate names-- may be more revealing. For example, it may be argued that the node **IndexedExtensibleCollection** enhances our understanding of what **Dictionary** and **LinkedList** do, and highlights a similarity between the two that would otherwise require more thorough navigation in the knowledge space. Second, in anticipation of future amendements/extensions to the class hierarchy, such a generalization may be worth keeping.

4. We developed an incremental algorithm that generates the knowledge space directly, rather than going through the Galois lattice first [Godin93a].

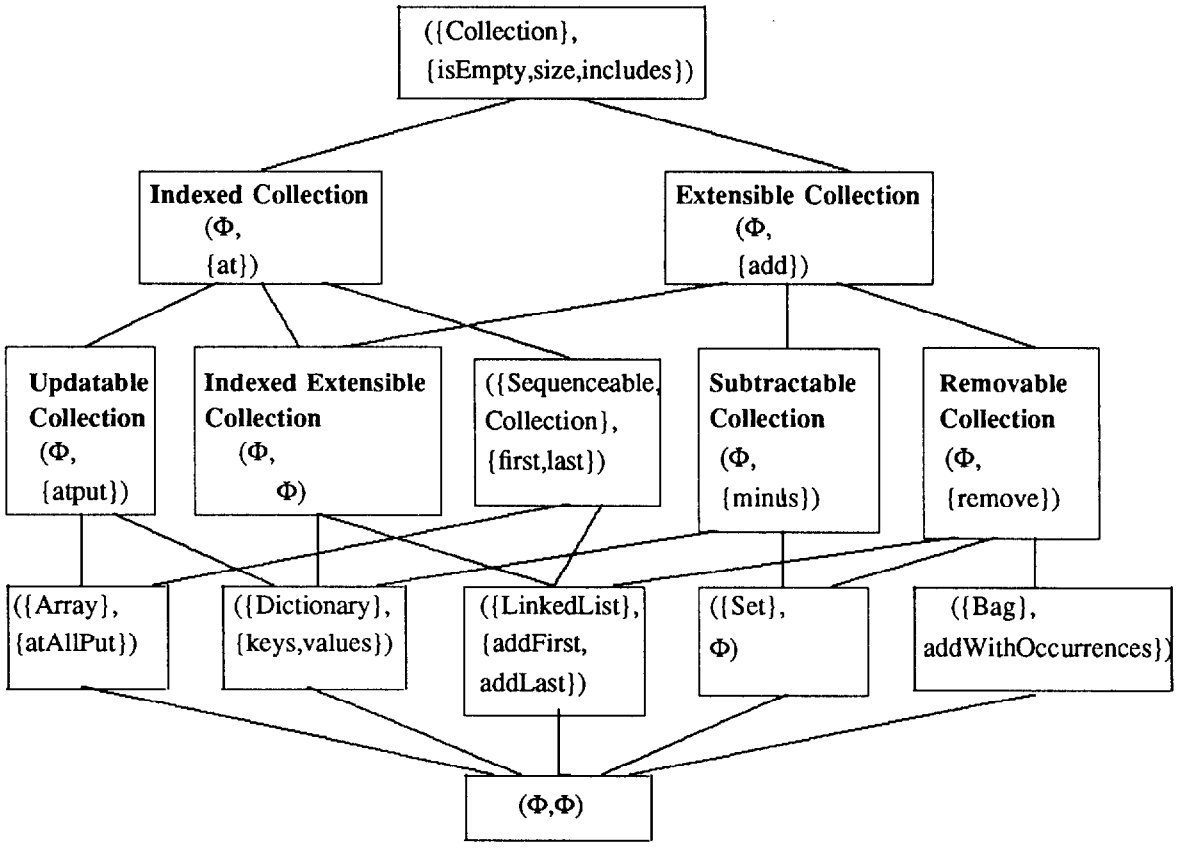


Figure 4. Inheritance Galois lattice with abstract class names

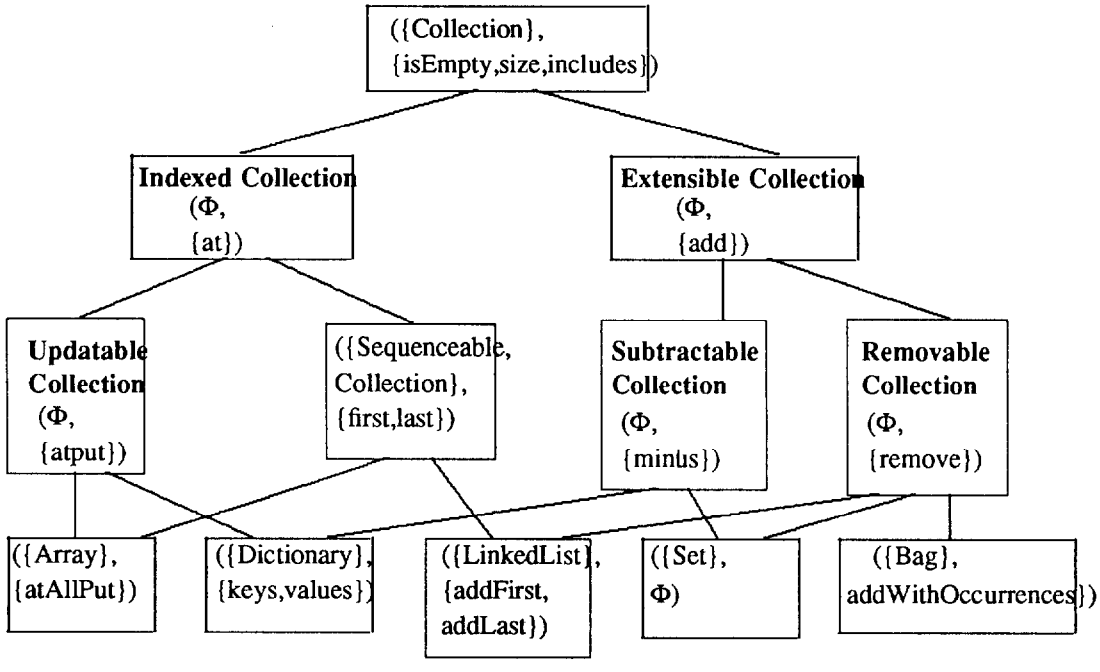


Figure 5. Inheritance knowledge space for the relation in Figure 1.

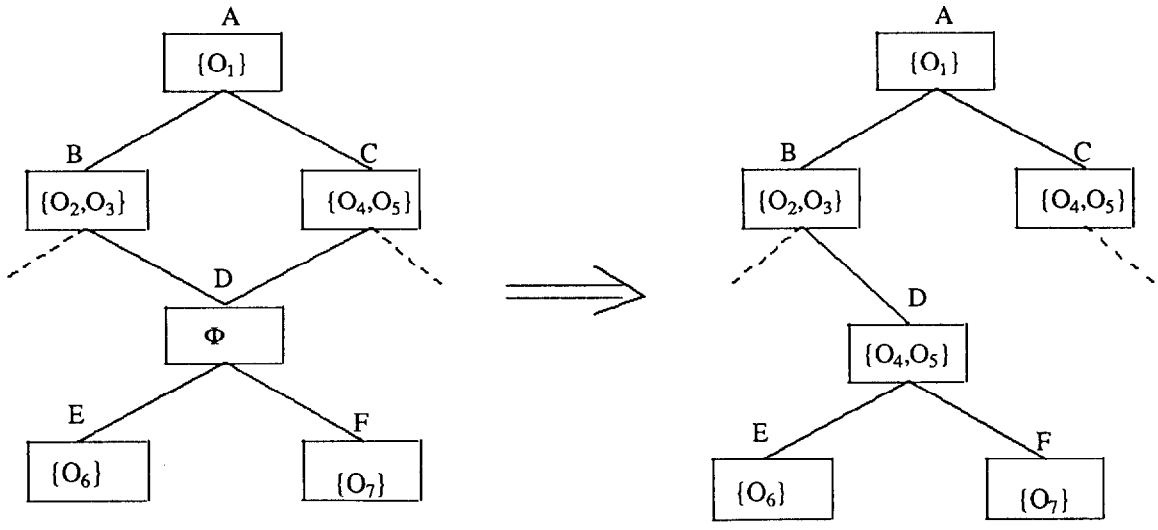


Figure 6-a. Implementing an inheritance Galois lattice with single inheritance

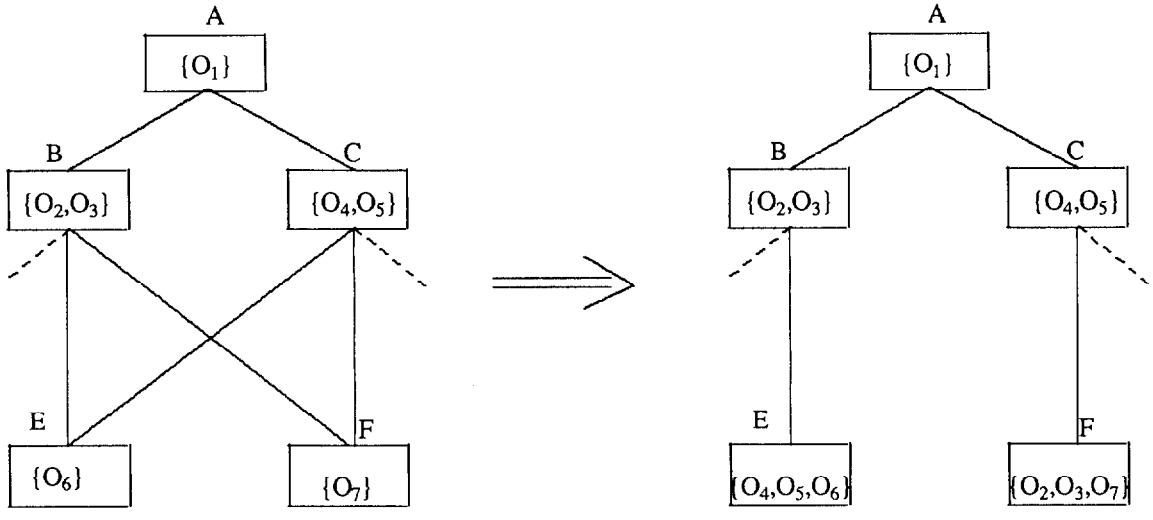


Figure 6-b. Implementing a knowledge space (KS) with single inheritance

Figure 6. Keeping (Φ,Φ) pairs reduces the amount of duplication during design.

Notice that neither justification would necessarily compel us to implement such a class: we could prescribe that the full inheritance Galois lattice be kept at the analysis level for the sake of understandability, but that it should be streamlined at the design stage.

As it turns out, design issues may actually be the most compelling reasons to keep such nodes. For instance, in languages that do not support multiple inheritance, some methods will have to be implemented at least twice. Such nodes may be required to reduce the amount of duplication. Figures 6-a and

6-b illustrate this point. In Figure 6-a, a decision was made to detach D from C, and reimplement C's methods (O_4 and O_5). Such a decision may be based on the (anticipated) relative complexity of $\{O_4,O_5\}$ with respect to $\{O_2,O_3\}$. In Figure 6-b, we showed one of three possibilities, each of which would result in reimplementing/duplicating 4 methods (or 2 methods twice).

3. Using Richer Class Descriptions

The description of classes used by the method presented in the previous section consisted simply of a list of items. The fact that these items corresponded to method names had no bearing on the technique. We could have used data attributes instead, or both. Some OOA methodologies suggest using associations as additional criteria for classification (see e.g. [Rumbaugh91a, Bergstein91a]). Using the appropriate notation, associations could also be included in the descriptions of classes, and used for classification[5]. However, the classification method-- as presented-- implicitly assumes straight inheritance along the paths of the lattice. In particular, it does not handle explicitly cases where methods may have to be redefined/refined at lower levels and *ultimately* require separate-- if only partial-- implementations. We cannot consider such methods as completely different, for otherwise we miss a useful abstraction. At the same time, they could not be considered equal, and some of the operations performed to derive the inheritance Galois lattice and knowledge space may no longer be justified. Actually, this is one instance of the more general problem where a partial order relation exists between descriptors themselves. We describe the changes needed to the basic method to take advantage of this relationship in discovering useful abstractions.

5. In fact, come implementation time, the three may not be distinguishable: 1) access methods are needed for both the stored (e.g. birthdate) and the computed (e.g. age) instance variables, 2) some computationally costly methods may store their results, and 3) some associations may be implemented as instance variables or as methods [Rumbaugh91a].

3.1. Using Partial Order Relations Between Class Descriptors

For the sake of illustration, we use an example from the procedural world where we take *multi-faceted descriptions* of software procedures [Prieto-Diaz87a]. For our purposes, the facets are only important to the extent that they define a domain of comparable values. In an actual class description problem, if operations are given by their signatures, operations with the same name will be compared using signature conformance relationships as in [Meyer88a]. If operations are given by formal relational specifications, then we may compare specifications whose domains have a non-empty intersection. And so forth.

Consider 5 software components, numbered 1 through 5, described by three properties (or *facets* [Prieto-Diaz87a]): 1) *Function*, describing the purpose of the procedure, 2) *Object*, indicating the main data object transformed/operated on by the procedure, and 3) *Medium*, description the medium in which the object resides while being manipulated. Figure 7 shows the values of the facets for the 5 components. The corresponding full Galois lattice is shown below (Figure 8).

	Function	Object	Medium
1	quicksort	real	array
2	heapsort	string	array
3	mergesort	string	file
4	print	string	file
5	search	integer	array

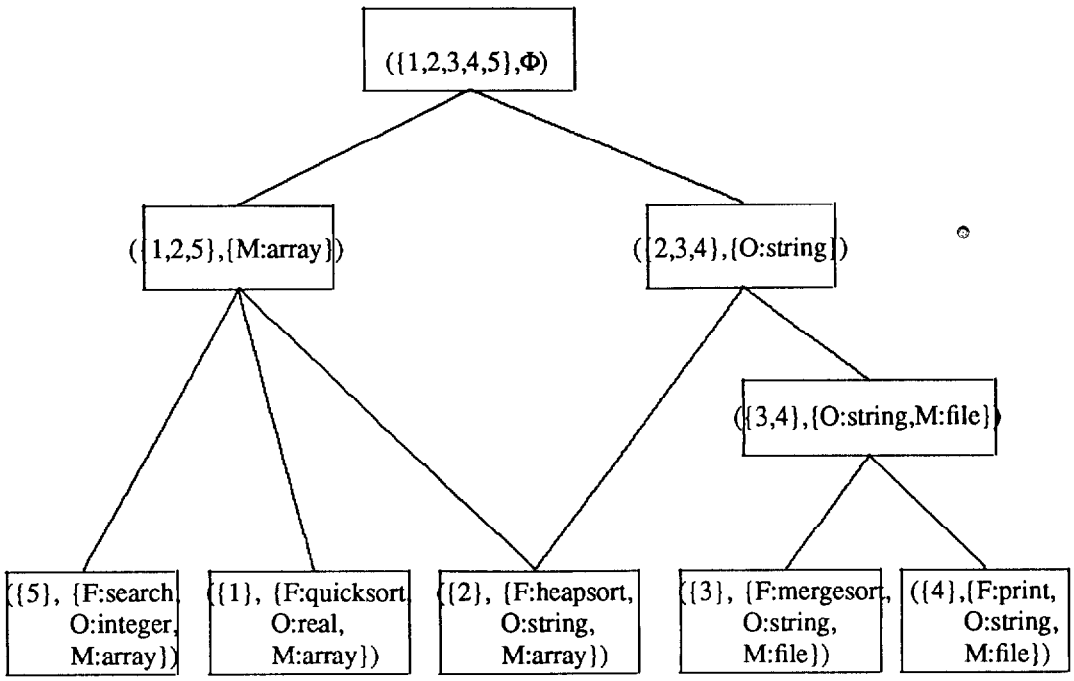


Figure 8. Full Galois lattice based on Figure 7. The "bottom" of the lattice (Φ, <union of leaf nodes>) is not shown.

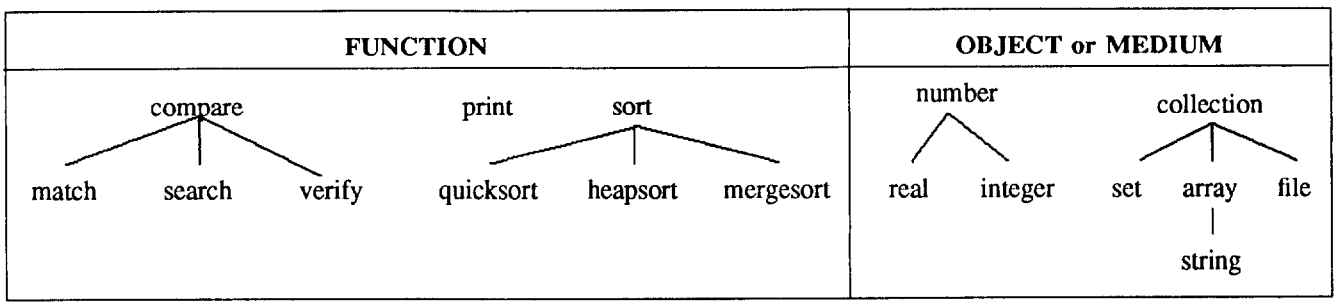


Figure 9. Partial order relations between facet values.

The lattice in Figure 8 shows a few useful abstractions. If we knew the relations between the values of the facets that are shown in Figure 9, we could identify a useful abstraction between components 1 and 2, which corresponds to sorting arrays. Further, both 1 and 5 operate on arrays of numbers. We define a partial order "<" between facet-value pairs, noted "facet:value", as follows:

$$(f_1{:}t \;<\; f_2{:}t') \;\longleftrightarrow\; [(f_1 = f_2) \wedge (t \;LT\; t')]$$

where LT is the partial order between values. Referring to Figure 9, we have: "Function:quicksort" < "Function:sort". By contrast, no relation exists between "Function:quicksort" and "Function:print", or between "Function:print" and "Medium:file". Using this definition, we induce from the relation R that relates a component to a "facet:value" pair, a new relation R+ defined as follows:

$$(x,x') \in R+ \equiv (\exists\, y')\; [(y' \le x') \wedge (\,(x,y') \in R\,)]$$

In other words x' is related to x by R+ if one of its descendants is related to x by R, i.e., one of its descendants is a facet:value pair for the component x. For example, (1, "Function:sort") belong to R+, because "Function:quicksort", a descendant of "Function:sort", is a facet:value pair of component 1.

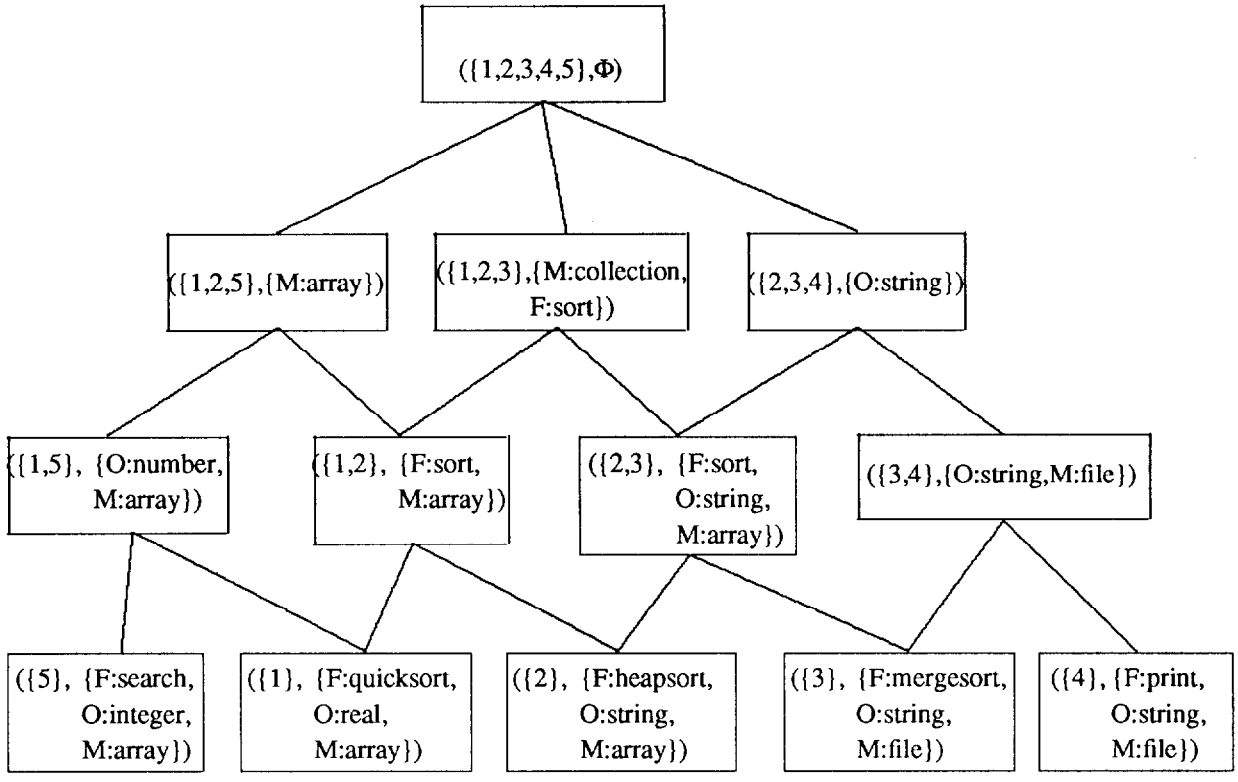


Figure 10. New lattice using partial order relations between facet values.

The new lattice consists of (X,X') pairs where:

1) $X' = f(X) = \{x' \in E' \mid (\forall\ x \in X), (x,x') \in R+)$, and $(\exists \neg\ y')$ s.t. $y' < x'$, and for all $x \in X$, $(x,y') \in R+\}$
2) $X = f'(X') = \{\ x \in E \mid \forall\ x' \in X', (x,x') \in R+\}$

With regard to f(X), notice that of all the elements of E' that are related (through R+) to every single element of x, we keep only the "smallest" ones. In particular, if all the elements of X have one facet:value pair in common, only that pair will appear in X', and none of its ancestors.

Figure 10 shows the new lattice. Compared to Figure 8, four new "abstractions" were created: {1,2,3}, to sort collections, {2,3}, to sort collections of strings, {1,2}, to sort arrays, and {1,5}, to operate on arrays of numbers. The reader may check that when we apply the reduction techniques of sections 2.2 and 2.3, only {1,2,3} survives in the final knowledge space.

3.2. Examples

3.2.1. Refining Instance Variables

Unlike methods, in most OOP languages, instance variables are inherited as is: they may not be redefined[6]. Most knowledge representation languages support a number of partial order relationships between concepts' attributes [Mili92a], among which *value restriction* and *specialization*. For example, the class **Person** may define "Age" as belonging to the interval 0..100. The subclass **Teenager** may *restrict the values* of "Age" to the interval 13..19. Similarly, a class **Organiza-**

6. In Eiffel, attributes may be *renamed* by subclasses [Meyer88a].

tionalUnit may have a property called "Leading Officer", and a subclass may refine that to "Chairperson", "Director", "Chief Executive Officer", etc, all of which are *specializations* of "Leading Officer". In knowledge representation languages, the same semantics would be used to represent specialization between properties (slots) as that between "concepts"[7] (see e.g. KL-ONE [Brachman85a]).

At the OO analysis level, we are typically dealing with application domain knowledge, the kind for which such nuances are significant to the user/analyst. Short of allowing for such a flexibility in modeling, and *providing the required application domain knowledge base to back it up*, we have to either: 1) push nomenclature standardization to ridiculous limits to make sure that common abstractions are properly factored out, or 2) forgo opportunities for useful generalizations. Comes the design and implementation level, value restriction can be enforced via a redefinition of the access methods. In the "Age" example, we simply redefine setAge() to reject arguments outside the [13..19] interval, reducing this case to the next (method conformance). As for specialization, we could rename both the instance variables and their access methods, *à la* Eiffel [Meyer88a], or, if the language does not support it, simply provide additional access methods within subclasses that would have more significant names.

3.2.2. Method Conformance

With methods, a range of partial orders may be defined, depending on the formalism used to specify the methods, and on the knowledge/tools available to the classification tool. If methods are specified simply by their-- presumably domain-meaningful-- names, an explicit taxonomy of domain tasks would need to be referenced by the classification tool to identify "conformance" relationships. Such a taxonomy might identify, e.g., "**computeWeeklyPay**" as a special case of "**computePay**". If methods are specified by their signatures, conformance may be defined by covariance. In this case, the classification tool tests methods that have the same name for conformance, and will only test those. The subtype/subclass relationships between method arguments could be either available explicitly, or computed. If pre-conditions and post-conditions are used, some sort of a theorem prover would be needed to establish implication relationships between pre/post-conditions[8]. Naturally, the richer the specification technique/language used, the more reliable are the generalizations, and the more complex is the process of establishing them.

It is worth pointing out that when conformance relations between methods have to be computed "on the fly", computing the pairs of sets (X,X') for the Galois lattice (more specifically, the set X', see § 3.1) involves computing the lowest upper bound of a set of methods. Interestingly, Mili et al. have developed an algorithm that organizes a set of relational specifications into a lattice [Mili92b]. Such a lattice would have to be made available to our Galois lattice generation algorithm before the organization of classes can proceed.

3.2.3. Associations

A number of OO development methodologies prescribe that associations (conceptual relationships) be both: 1) used as criteria for generalization (to the same degree that attributes and methods do), and 2) be subject to generalization (see e.g. [Wirfs-Brock90a, Rumbaugh91a, Bergstein91a]). We wish to qualify the first statement by restricting this to what we call *defining associations* such as aggregation, as in [Bergstein91a], associations involving

7. with interesting implications, such as the circularity and hence undecidability of classification!

8. In Eiffel, the "covariance" of pre/post-conditions, if we may call it that way, does not require proving because of the way the compiler treats the pre-conditions (**require** clause) and post-conditions (**ensure** clause): 1) pre-conditions are implicitly ORed with those from the inherited version, and 2) post-conditions are implicitly ANDed with those from the inherited version [Meyer88a]. Such a strategy would not work in our case, because we don't know that the method redefines/refines an inherited version; that is what we try to establish!

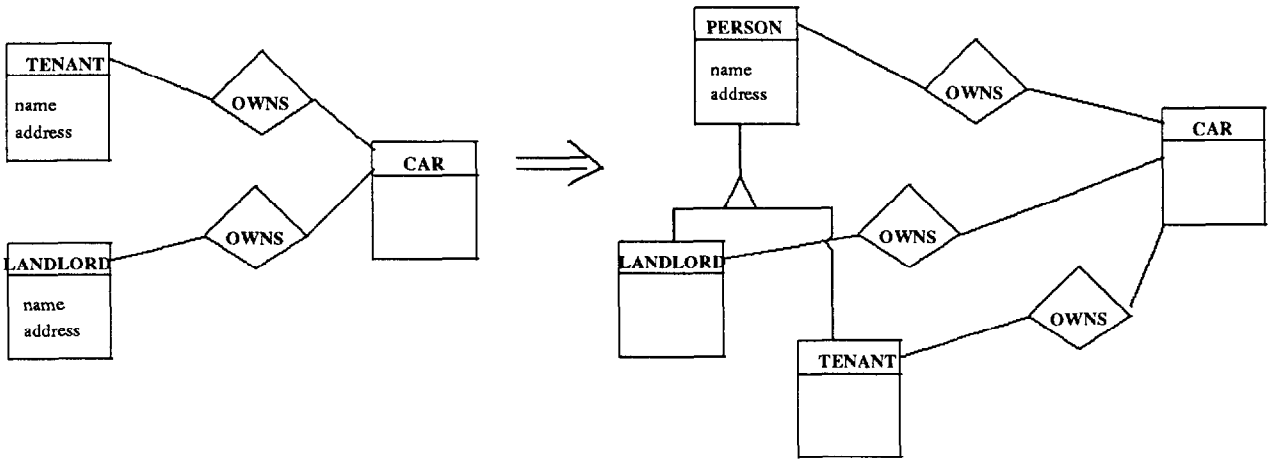


Figure 11. Generalizing conceptual graphs.

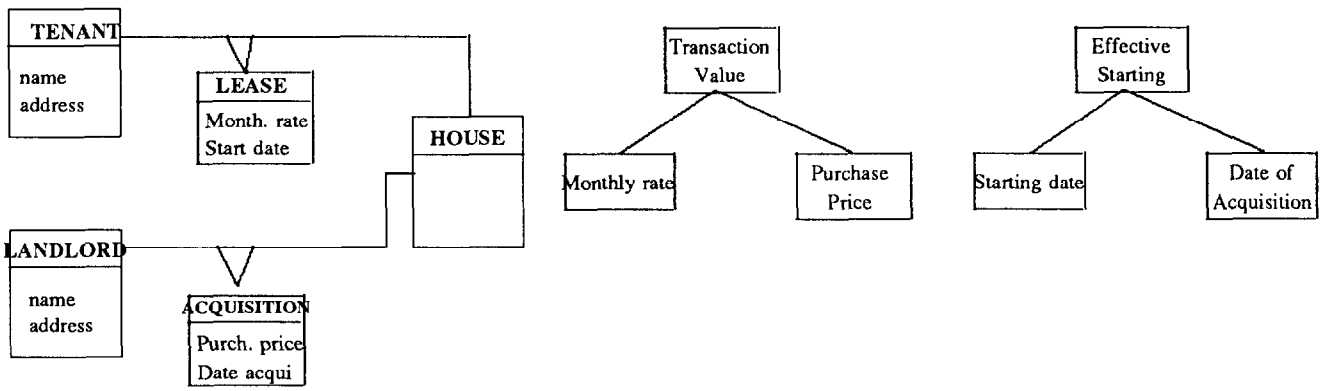
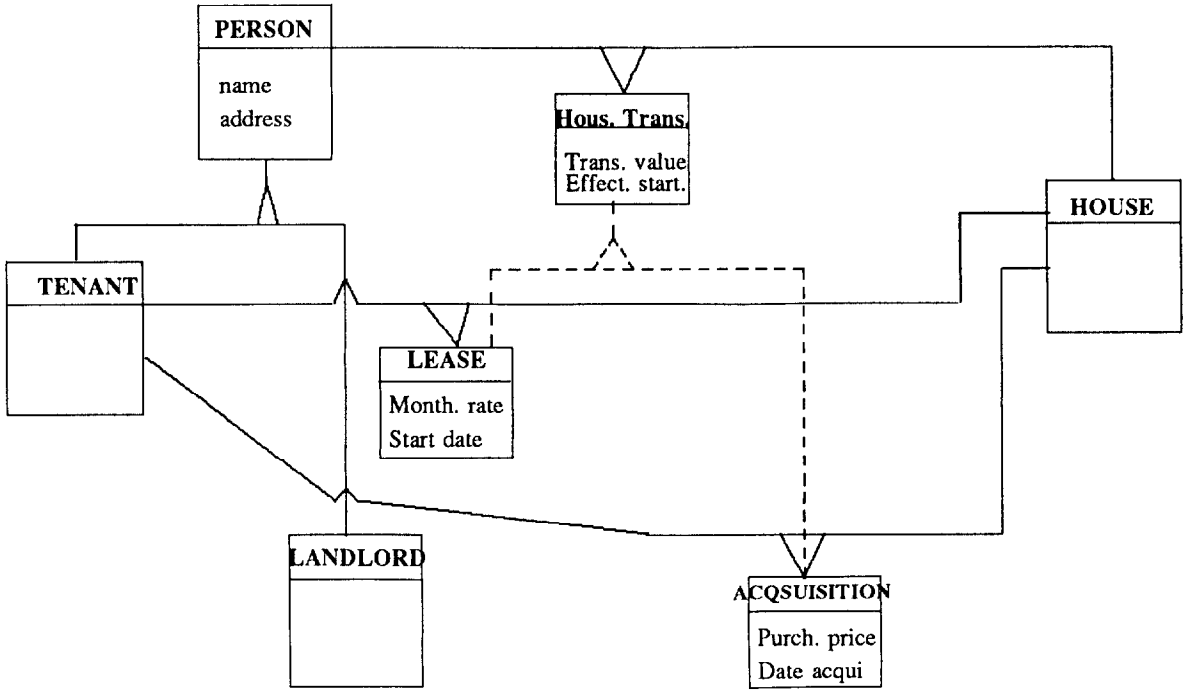


Figure 12. A case of "circular" generalization.

existential dependencies, and *intrinsic constraints* [Mili90a]. In one application, the method presented in section 2 has been extended to handle concepts represented as conceptual graphs [Mineau90a]. Such an algorithm is capable of generating the generalization shown in Figure 11. Notice that in the resulting "lattice", all three associations are kept. An "inheritance lattice" of conceptual graphs [Godin93a] would eliminate the "redundant" OWNS associations between **Tenant** and **Landlord** on one hand, and **Car** in the other.

Consider now the trickier case of Figure 12, where we used Rumbaugh's object model notation. The target generalization is shown in Figure 13. However, such a generalization cannot systematically be obtained from the method shown in § 3.1. For instance, while the similarity between **Tenant** and **Landlord** is sufficient to suggest the parent concept **Person**, the common generalizations between the attributes of the associations (see [Rumbaugh91a]) are not sufficient to suggest the generalization of the **Lease** and **Acquisition** associations to the **Housing Transaction** association: associations are first and foremost defined by the classes they relate. Thus, *prior* to establishing that **Tenant** and **Landlord** *generalize to* **Person**, *we cannot say that* **Lease** and **Acquisition** generalize to **Housing Transaction**.



This is another instance of the more general problem of "circular" classification, often encountered in knowledge representation languages [Lipkis82a]: classification is expressed in terms of equivalence of logical expressions whose truth values can be established through classification. The consequences of such "circularity" go from undecidability to unpredictability. In this case, if the lattice generation method happens to generalize **Tenant** and **Landlord** first, then **Lease** and **Acquisition** will be properly generalized. A more serious problem occurs if we could not generalize **Tenant** and **Landlord** without generalizing the corresponding associations[9]. We had to amend the algorithm for conceptual graphs by "revisiting" some nodes of the lattices as new generalizations are found. In this case, we would have a lattice of associations where the related classes are used as features, and **Tenant** and **Landlord** would appear within the X' component of (X,X') pairs. Upon "discovering" the **Person** generalization, we revisit the nodes of the associations lattice whose X' component contain either **Tenant** or **Landlord**, and we recompute the nodes from that point upward using the algorithm for incremental updates [Godin93a].

9. This is only a problem in knowledge representation languages that do not allow for exceptions. In our case, we take the best generalizations we can get!

4. Discussion

In this paper, we proposed a formal method for building and maintaining hierarchies of class descriptions. The method is based on the concept of Galois lattice of a binary relation and refinements thereof. One such refinement, the inheritance knowledge space, relates directly to the notion of protocol hierarchy described in [Cook92a]. Our method has some advantages compared to other conceptual clustering methods [Godin86a]:

1) it supports an efficient incremental update algorithm,
2) it does not depend on any (subjective) parameter tuning or input ordering, and
3) the resulting hierarchy is not limited to be a tree.

Over the years, we developed a number of extensions to the basic method to take into account richer class descriptions [Mineau90a]. We showed how

these extensions may be advantageously used to provide greater flexibility in OO analysis, and showed how the enhanced models can be mapped to design and implementation.

We are currently involved in a major government-industry collaboration on OO methodologies for distributed systems. The ideas presented in this paper are being implemented as part of an OO development toolkit, and will be tested on distributed systems software (DSS). Given the higher-than-average use of formal specification techniques in DSS, we expect the project to provide a good testbed for the range of methods discussed in this paper. We are also exploring an adaptation of the method to the case where the application domain represents several functional views of the same data objects. We believe that the views may have to be modeled separately. Further, we see distinct advantages to *implementing distinct class hierarchies that delegate to the same base classes*, which act as common datastores [Mili92c]. One of the research challenges consists of detecting these views *a-posteriori*, from the "anatomy" of the lattice, and suggesting a subdivision of class descriptions, each corresponding to a functional view; In other words, we are concerned with developing a slicing algorithm for Galois lattices that maximizes some measure of cohesion within the "slices" [Mili92c]. In some cases, this approach would obviate the need for-- the problematic-- multiple inheritance. In general, we believe that this approach leads to flexible, yet "clean" (conformant, no-cancellations, etc) class hierarchies.

Acknowledgements. Robert Godin and Hafedh Mili are supported through individual operating grants from Canada's Natural Sciences and Engineering Research Council (NSERC), and a team grant funded through Quebec's SYNERGIE collaborative research initiative.

References

Bergstein91a.
P. Bergstein and K. J. Lieberherr, "Incremental class dictionary learning and optimization," in *Proceedings of ECOOP'91*, ed. Springer Verlag, pp. 377-395, Geneva, Switzerland, 1991.

Brachman85a.
Ronald J. Brachman and James G. Schmolze, "An Overview of the KL-ONE Knowledge Representation System," *Cognitive Science*, vol. 9, pp. 171-216, 1985.

Coad91a.
Peter Coad and Edward Yourdon, in *Object-Oriented Analysis*, Prentice Hall, 1991. second edition

Cook92a.
William R. Cook, "Interfaces and Specifications for the Smalltalk-80 Collection Classes," in *Proceedings of OOPSLA'92*, pp. 1-15, ACM Press, Vancouver, B.C., Canada, October 18-22, 1992.

Cox90a.
Brad J. Cox, "Planning the Software Revolution," *IEEE Software*, vol. 7(6), pp. 25-35, November 1990.

Ganter86a.
B. Ganter, J. Stahl, and R. Wille, "Conceptual Measurement and Many-Valued Contexts," in *Classification as a Tool of Research*, ed. M. Schader, pp. 169-176, North-Holland, Amsterdam, 1986.

Gennari90a.
J. H. Gennari, P. Langley, and D. Fisher, "Models of Incremental Concept Formation," in *Machine Learning: Paradigms and Methods*, ed. J. Carbonell, pp. 11-62, MIT Press, Amsterdam, the Netherlands, 1990.

Gibbs90a.
S. Gibbs, D Tsichritzis, E. Casais, O. Nierstrasz, and X. Pintado, "Class Management for Software Communities," *Communications of the ACM*, vol. 33, no. 9, pp. 90-103, 1990.

Godin93a.
Robert Godin, Rokia Missaoui, and Alain April, "Experimental Comparison of Navigation in a Galois Lattice with Conventional Information Retrieval Methods," *International*

Journal of Man-Machine Studies, 1993. To appear.

Godin86a.
R. Godin, E. Saunders, and J. Gecsei, "Lattice Models of Browsable Data Spaces," *Journal of Information Sciences*, vol. 40, pp. 89-116, 1986.

Godin91a.
R. Godin, R. Missaoui, and H. Alaoui, "Learning Algorithms Using a Galois Lattice Structure," in *Proceedings of the Third International Conference on Tools for Artificial Intelligence*, pp. 22-29, IEEE Computer Society Press, San Jose, CA, 1991.

Li89a.Q. Li and D. McLeod, "Object Flavor Evolution in an Object-Oriented Database System," in *Proceedings of the Second International Conference on Expert Database Systems*, ed. L. Kershberg, pp. 469-495, Benjamin/Cummings, 1989.

Lipkis82a.
Thomas A. Lipkis, "A KL-ONE Classifier," in *Proceedings of the 1981 KL-ONE Workshop*, ed. J.G. Schmolze & R. J. Brachman, pp. 128-145, Bolt Beraneck and Newman, Inc., June, 1982.

Meyer88a.
Bertrand Meyer, in *Object-Oriented Software Construction*, ed. Prentice-Hall International, 1988.

Mili92b.
Ali Mili, Noureddine Boudrigua, and Fathi Elloumi, "The Lattice of Specifications: Applications to a Specification Methodology," *Formal Aspects of Computing*, Springer-Verlag, 1992. to appear

Mili90a.
Hafedh Mili, John Sibert, and Yoav Intrator, "An Object-Oriented Model Based on Relations," *Journal of Systems and Software*, vol. 12, pp. 139-155, 1990.

Mili92a.
Hafedh Mili and Roy Rada, "A Model of Hierarchies Based on Graph Homomorphisms," *Computers and Mathematics with Applications*, vol. 23, no. 2-5, pp. 343-361, Winter 1992.

Mili92c.
Hafedh Mili and Robert Godin, *Software Reuse Research Plan for the SYNERGIE Project*, p. 16, Department of Maths and Computer Science, Univ. of Quebec at Montreal, November 17, 1992.

Mineau90a.
G. Mineau, J. Gecsei, and R. Godin, "Structuring Knowledge Bases using Automatic Learning Processes," in *Proceedings of the Sixth International Conference on Data Engineering*, pp. 274-280, IEEE Computer Society Press, Los Angeles, CA, 1990.

Ossher92a.
Harold Ossher and William Harrison, "Combination of Inheritance Hierarchies," *SIGPLAN Notices*, vol. 27, no. 10, pp. 25-40, Vancouver, B.C. (Canada), October 18-22, 1992. Proceedings of OOPSLA'92.

Pedersen89a.
Claus H. Pedersen, "Extending Ordinary Inheritance Schemes To Include Generalization," *SIGPLAN Notices*, vol. 24, no. 10, pp. 407-418, New Orleans, Louisiana, October 1-6, 1989. Proceedings of OOPSLA'89.

Prieto-Diaz87a.
Ruben Prieto-Diaz and Peter Freeman, "Classifying Software for Reusability," *IEEE Software*, pp. 6-16, January 1987.

Rumbaugh91a.
James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, in *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

Wille82a.
R. Wille, "Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts," in *Ordered Sets*, ed. I. Rival, pp. 445-470, Reidel, Dordrecht-Boston, 1982.

Wirfs-Brock90a.
Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener, in *Designing Object-Oriented Software*, Prentice-Hall, 1990.