Design of class hierarchies based on concept (Galois) lattices

Robert Godin[1], Hafedh Mili[1], Guy W. Mineau[2], Rokia Missaoui[1], Amina Arfi[1], Thuy-Tien Chau[1]

1 - Département d'Informatique
Université du Québec à Montréal
C.P.8888, Succursale Centre Ville
Montréal, Québec
Canada, H3C 3P8
(514) 987-3088
Fax: (514) 987-8477
(godin.robert@uqam.ca)

2- Département d'Informatique
Faculté des Sciences et de Génie
Université Laval
Québec, Québec
Canada, G1K 7P4
(418) 656-5189
Fax: (418) 656-2324
(mineau@ift.ulaval.ca)

Abstract

Building and maintaining the class hierarchy has been recognized as an important but one of the most difficult activities of object-oriented design. Concept (or Galois) lattices and variant structures are presented as a framework for dealing with the design and maintenance of class hierarchies. Because the design of class hierarchies is inherently an iterative and incremental process, we designed incremental algorithms that update existing Galois lattices as the result of adding, removing, or modifying class specifications. A prototype tool incorporating this and other algorithms has been developed as part of the IGLOO project, which is a large object-oriented software engineering joint research project involving academic and industrial partners. The tool can generate either the concept lattice or several variant structures incrementally by incorporating new classes one by one. The resulting hierarchies can be interactively explored and refined using a graphical browser. In addition, several metrics are computed to help evaluating the quality of the hierarchies. Experiments are presented to better assess the applicability of the approach.

1. Introduction

1.1 Overview

Building and maintaining the class hierarchy has been recognized as an important and one of the most difficult activities of object-oriented design (Booch, 1994). 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. For example the Object Behavior Analysis method proposed by Rubin and Goldberg (1992) involves a sub-step where after identifying the objects, their attributes and services, the class hierarchy is reorganized in order to represent relevant abstractions based on common behaviors and attributes. At the design level, such hierarchies are augmented and possibly reorganized in order to take into account the solution objects along with the application domain objects (Monarchi & Puhr, 1992). Several guidelines have been proposed for the design of class hierarchies. Among these, two characteristics emerge from the literature that are particularly relevant to this work.

1. *Minimizing redundancy*. There is a large consensus that keeping each thing in one place in the class hierarchy is a good software engineering practice (Casais, 1991; Johnson & Foote, 1988; Korson & McGregor, 1992; Lieberherr, Bergstein & Silva-Lepe, 1991). Having the same thing defined in many places may lead to inconsistencies between the duplicates often because of modifications made in the maintenance process. When making modifications, people might not be aware of the duplication. Redundancy may also mean that the right abstractions have not been identified based on commonalities within the library. Lessons from building large class libraries (Meyer, 1990) show that it is hard to identify good abstractions a priori and it is often necessary to reorganize a library to reflect the undetected commonalities.

2. *Subclasses as specializations*. Inheritance is sometimes used only for code reuse purposes, which, as observed by Cox (Cox, 1990), produces libraries that are difficult to understand and to reuse. Many authors advocate that the inheritance hierarchy be as consistent as possible with specialization (Casais, 1991; Coleman, Arnold, Bodoff, Dollin, Gilchrist, Hayes, et al., 1994; Cook, 1992; Johnson & Foote,

1988; Lalonde, 1989; Liskov, 1988) to achieve better understandability and reusability. In addition, the correctness of polymorphic substitution of an instance of a subclass for an instance of a superclass depends on this approach to inheritance.

Initially proposed for knowledge acquisition and discovery in the artificial intelligence field (Godin & Missaoui, 1994; Godin, Missaoui & Alaoui, 1991; Wille, 1982; Wille, 1992), concept (or Galois) lattices have recently found interesting applications in software engineering (Godin, Mineau, Missaoui, St-Germain & Faraj, 1995a; Krone & Snelting, 1994). In (Godin & Mili, 1993) concept lattices and derived structures were proposed as a framework for dealing with the construction and maintenance of class hierarchies. Within that framework, hierarchies were guaranteed to have zero redundancy, by factoring out commonalities, and to be consistent with specialization. Another advantage of the theory is that it provides a clear and simple definition of the nature of the generated hierarchy that does not depend on algorithm specifics or parameter tuning, as opposed to most work in concept formation methods (Fisher & Pazzani, 1991; Gennari, Langley & Fisher, 1990) and other algorithms for dealing with class hierarchies (Casais, 1991; Dvorak, 1994; Lieberherr et al., 1991). Because the design of class hierarchies is typically an iterative and incremental process, we developed efficient incremental algorithms that update the hierarchies by adding, removing, or modifying the specifications of classes.

Notice that under the constraints of maximal factorization (everything defined only once) and consistency with specialization, a given set of class specifications may yield several class hierarchies-- including the Galois concept lattice and its variants. Notwithstanding language differences (e.g. support for multiple inheritance), we have been interested in comparing those solutions from the point of view of complexity and maintainability. To this end, we designed a set of metrics that build upon an increasing body of work on structural metrics for object-oriented software (Lieberherr et al., 1991, Chidamber & Kemerer, 1994). We are interested in measuring *redundancy*, *complexity* as reflected by the number of specialization and aggregation links, and *deviation from specialization*. These measures should enable us to compare the variants of concept lattices among themselves, and to other manually or automatically-built class hierarchies.

Using the above framework of concept lattices and related algorithms for class hierarchies, a tool has been developed using ObjectWorks Smalltalk, within the context of the *IGLOO*[1] project, a large joint research effort involving several Québec universities and Québec-based companies, addressing various aspects of object-oriented software engineering. The tool can generate the concept lattice, as well as several variant structures, incrementally by incorporating new classes one by one. The hierarchies can be explored using a graphical browser. Further, several structural metrics are computed to help evaluate the produced hierarchies.

We performed two sets of experiments to test the effectiveness of our approach. The first set of experiments was applied to the classes in the **`Collection`** sub-hierarchy of the ObjectWorks\Smalltalk class library. The purpose of the experiment was twofold : a) validating, empirically, the complexity of the lattice incremental construction algorithms, and b) applying the structural metrics to the Galois lattice and its variants, as well as to the original class hierarchy. Extracting class interfaces was, in and of itself, a non trivial problem because of the various inheritance cancellation mechanisms used in the Smalltalk library. Overall, experiments validated the theoretical complexity of the incremental algorithms, and showed the superiority of the Galois lattices, as compared to the original class hierarchy, with regard to the structural metrics, Smalltalk's single inheritance notwithstanding. They also showed that the three quality criteria (redundancy, complexity, deviation from specialization) are somewhat contradictory, that the different variants of Galois lattices strike different trade-offs between them, and illustrating once more the difficulty of class hierarchy design.

The second set of experiments aimed at validating the approach for a small set of fairly large class specifications, as is often the case with domain models. To this end, we applied the algorithm to a *management information base* (MIB), which is a set of class specifications pertaining to telecommunication network management functionalities. The experiments showed that the Galois lattice

[1] IGLOO stands for InGénierie du Logiciel Orienté Objet.

helped identify non-trivial generalizations, but at the same time, suffered from the lack of aggregation as a concept formation construct, as is the case with the Demeter system (Lieberherr et al., 1991).

Next, we discuss alternative approaches to automatic class organization and re-organization, and highlight ways in which our approach is superior, or more easily applicable than the existing body of work. In section 2, we present the basic definitions of concept lattices and of several variant structures useful for class hierarchies. Section 3 presents our structural metrics, along with motivations and examples. Section 4 describes the implementation and some experimental results. We conclude in section 5.

1.2 Litterature review

The problem of building an initial class hierarchy from a set of class specifications, or re-organizing an existing one following class updates, has been receiving increasing attention in the OO literature. Work in the context of the Demeter System has addressed the automatic discovery of class hierarchies from example objects (Lieberherr et al., 1991). They use a *class dictionary graph* to represent the design. Their algorithm uses a metric which underlies an optimization process. They propose a two-step learning algorithm where the first step does basic learning by generating a potentially non-optimal class dictionary graph. The edges of the graph represent the inheritance (alternation edges) and part-of (construction edges) relationships between classes. The second step optimizes the graph by trying to minimize a weighted function of the edges where the weight of the construction edges is at least twice the weight of the alternation edges. This function can be considered as a complexity metric for the class hierarchy. The metrics we have implemented in our tool are in part based on this work although we do not use them to guide the hierarchy construction process. Lieberherr et al. show that the exact optimization is NP hard and propose an approximate algorithm that brings down the complexity to a polynomial one. They also present an incremental algorithm that produces the optimum when it is a tree (Bergstein & Lieberherr, 1991). The same goals can be achieved within our framework with an important advantage: the resulting structure is clearly defined independently of any algorithm for producing it. A designer may be more inclined to use a tool where he could precisely understand why a given abstraction has emerged. The

incremental algorithms we use produce exactly the same result as the batch algorithms when applied to the same set of classes. Also, their algorithm does not address the possibility of a taxonomic relationship between properties.

Cook (1992) was more interested in critiquing and re-organizing existing class hierarchies to bring class (implementation) hierarchies as close as possible—language permitting—to type (interface) hierarchies. In one experiment, he automatically extracted interface specifications of the Smalltalk-80 **`Collection`** class library to build the corresponding interface hierarchy. He examined the produced hierarchy to detect problems with the actual library and proposed some improvements which are feasible within the context of Smalltalk's single inheritance. He also sketched a simple algorithm to generate the hierarchy. His algorithm is not incremental and the hierarchy has to be regenerated from scratch if any modification is done to the specifications. We show in section 2 how his structure is related to our formal framework. We also extend this work by showing other variant structures that can be relevant to the design of class hierarchies. In particular, we consider taking into account taxonomic relationships between properties. For example, our framework can handle overloading through type conformant redefinitions of operations.

Casais (Casais, 1991) proposed an approach that has essentially the same goals as ours, and that includes an incremental algorithm. However, his incremental algorithm relies on the user to specify the immediate superclasses of the new class, and it starts from that point and locally reorganizes the hierarchy. Consequently, the hierarchy is not guaranteed to be globally optimal and non redundant. Although he allows an interface to have several implementations and takes this into account in the process, he does not address the possibility of using specialization relationships between different implementations.

The ARES algorithm proposed in (Dicky, Dony, Huchard & Libourel, 1994) is also incremental and can generate the Galois subhierarchy from the class specifications. In (Dicky, Dony, Huchard & Libourel, 1996), Dicky et al. extend their work to handle overloading of properties. One major disadvantage of their algorithm is that it goes through every node of the hierarchy in a particular order. Our algorithm

only looks at a limited subset of the nodes. An advantage of their algorithm is that it can be used on hierarchies that are not necessarily maximally factorized.

Based on an empirical study, Dvorak (Dvorak, 1994) showed how class hierarchies tend to exhibit conceptual entropy which is manifested by increasing conceptual inconsistency as we travel down the hierarchy. As a solution, he proposed a method and an algorithm for building class hierarchies emphasizing conceptual simplicity and consistency. The algorithm relies on manually-generated formal specifications of the conceptual attributes for each class. The approach is also different from the preceding ones because the algorithm places the input classes in a tree structure without trying to discover new emerging abstractions based on observed commonalities. The framework and algorithms we propose could also be applied to the formal specifications when available.

Notice that the focus of these algorithms is how to put initially provided classes in a hierarchy and they do not address the restructuring of the basic objects and classes. There might be better alternative designs of these initial classes based on splitting or combining classes for example. Work on such behavior preserving transformations includes (Bergstein, 1991; Opdyke & Johnson, 1989). From the perspective of database schema evolution, Li and McLeod (Li & McLeod, 1994) have studied related problems in the handling of changes to the semantics of existing objects and classes.

2. Concept lattice and variants

In this section, we start with the basic definition of concept lattices. Then several variant structures are defined.

2.1 Concept lattice

We present the basic definition of a concept (Galois) lattice for a binary relation and illustrate it using a simple interface hierarchy example. Later, we extend the framework to handle more complex class specifications. More details about the underlying theory of concept lattices can be found in (Barbut & Monjardet, 1970; Davey & Priestley, 1992; Wille, 1992).

A (formal) context is a triple (G, M, I) where G and M are two finite sets and I is a binary relation between G and M, i.e. $I \subseteq G \times M$. The notation gIm is used instead of writing $(g,m) \in I$. Given $A \subseteq G$ and $B \subseteq M$, define:

$A' = \{m \in M \mid (\forall g \in A)\ gIm\}$ and

$B' = \{g \in G \mid (\forall m \in B)\ gIm\}$.

A *concept* of the context (G, M, I) is defined as a pair (A, B) where:

$A \subseteq G$, $B \subseteq M$, $A'=B$ and $B'=A$.

A partial order on the concepts can be defined as follows:

$(A_1, B_1) \leq (A_2, B_2)$ if $A_1 \subseteq A_2$ (which is equivalent to $B_2 \subseteq B_1$).

The set of all concepts for the context (G, M, I) with partial order $\leq$ is a complete lattice called the *concept* (or Galois) *lattice* of the context (Davey & Priestley, 1992) and is denoted here by $CL(G, M, I)$.

The set G is usually described as a set of objects and M as a set of attributes. For our purposes, the set G represents a set of *classes* and the set M represents a set of class *properties* including attributes (instance variables) *and* methods. Figure 2.1.1 shows an example context represented as a Boolean matrix. In the example, only message selector names appear as properties although we will consider other possibilities

later. The example in Figure 2.1.1 was extracted from (Cook, 1992) and represents a subset of the interfaces for a subset of the Collection classes in the ObjectWorks Smalltalk class library. This example is used for illustration purposes. For the classes appearing in the example, a sufficient subset of methods was chosen in order to relate the resulting structure to that given in (Cook, 1992). A message selector is related to a class by the binary relation if it is part of the *interface* of the class, i.e. the set of legal messages to which that the class can respond without returning an error. More details on the interface extraction are given in section 3. The exact syntax of a message selector is not respected in the example. The names given here are the concatenation of the argument keywords for each selector.

	Set	Bag	Dictionary	Linked List	Array
isEmpty	1	1	1	1	1
size	1	1	1	1	1
includes	1	1	1	1	1
add	1	1	1	1	0
remove	1	1	0	1	0
minus	1	0	1	0	0
addWithOccurrences	0	1	0	0	0
at	0	0	1	1	1
atput	0	0	1	0	1
atAllPut	0	0	0	0	1
first	0	0	0	1	1
last	0	0	0	1	1
addFirst	0	0	0	1	0
addLast	0	0	0	1	0
keys	0	0	1	0	0
values	0	0	1	0	0

Figure 2.1.1. Matrix representation of a context.

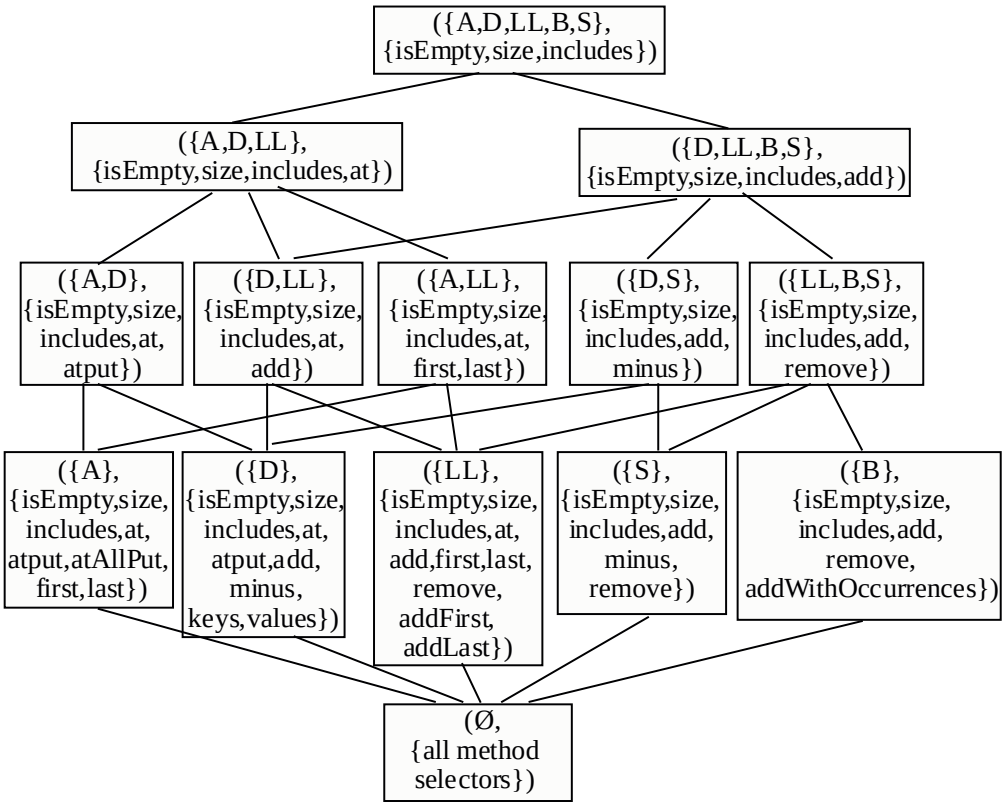

Figure 2.1.2. Concept lattice for the context in Figure 2.1.1.

Figure 2.1.2 shows the corresponding concept lattice. Only the uppercase letters for the class names appear in the Figure. The partial order is used to generate the graph in the following manner: there is an edge (C_1, C_2) if $C_1 < C_2$ and there is no other element C_3 in the lattice such that $C_1 < C_3 < C_2$. C_1 is called *parent* of C_2 and C_2 *child* of C_1. The graph is usually called a *Hasse diagram*. When drawing a Hasse diagram, the edge direction is implicit (here upwards).

The concept lattice reveals the commonalities between the classes of the context. The first part of the concept is the set of classes and the second part shows their common properties. This factorization of common properties can therefore be used in the class hierarchy design process. The Hasse diagram shows the generalization/specialization relationship between the concepts corresponding to the subset relationship between the property and objet sets. Therefore the graph can be used to produce hierarchies which are consistent with specialization. A fundamental property of *CL* is that it is a complete lattice

since for any subset of *CL* there exists a unique greatest lower bound and a unique least upper bound (Davey & Priestley, 1992).

The concept lattice generated from the interfaces in such a manner is closely related to the interface hierarchy defined in (Cook, 1992). The are some differences, which are explained below. In the case of class protocols, each concept represents a set of classes with their common interface and the Hasse diagram corresponds to the conformance relationship as defined by the inclusion of the sets of message selectors of the interfaces. The explicit representation of the sharing among interfaces is used as a tool for analyzing a library in (Cook, 1992). The interface hierarchy may also be useful to a reuser for browsing from a perspective which is different from the class (implementation) hierarchy and that is closer to the clients' view of the library (type hierarchy). This is similar in spirit to the work of (Oosthuizen, Bekker & Avenant, 1992). In this paper, we go futher by showing how to take into account different, but type conformant (i.e. partially-ordered) versions of the same method

It is important to note that taking all possible subsets produces an exponential number of concepts. However, when there is a fixed upper bound on $||\{g\}'||$,

$$||\{g\}'|| \leq K, \forall g \in G,$$

which is usually the case in practical applications, the worst case complexity of the structure is linearly bounded with respect to $||G|| = n$ (Godin, Saunders & Gecsei, 1986):

$$||\mathrm{CL}|| \leq 2^K n.$$

The upper bound on ||CL|| is exponential in *K*; however, experience with real applications and theoretical results with randomly assigned elements show that in practice, $||\mathrm{CL}||/n$ is fairly stable and much smaller than this upper bound (Carpineto & Romano, 1996; Godin, Missaoui & April, 1993).

Another important factor for the practical use of the lattice is the existence of incremental algorithms for updating the lattice structures (Carpineto & Romano, 1996; Godin, Missaoui & Alaoui, 1995d). Empirical data from several applications showed that new elements can be added in $\mathbf{O}(n)$ time (Godin et al., 1995d). Under the hypothesis of a fixed upper bound on $||\{g\}'||$, this is also confirmed by a

complexity analysis. This upper bound hypothesis is a reasonable one for class features, although the upper bound might be large for the case of message selectors, as opposed to instance variables.

2.2 Inheritance concept lattice

The concept lattice itself is not adequate for class hierarchies because of the redundancy in the representation. For a pair $C = (A, B)$, A will be present in every ancestor of C and symmetrically, B will appear in every descendant. For the class hierarchy design problem, the duplication should therefore be eliminated and be computed when needed by the inheritance mechanism. For a concept $C = (A, B)$, let AN be the non-redundant elements in A, and BN the non redundant elements in B. A class g will appear in AN if the corresponding concept C is the greatest lower bound of all concepts containing g. Symetrically, a property m will appear in BN if the corresponding concept C is the least upper bound of all concepts containing m. The lattice property guarantees the existence of greatest lower bound and the least upper bound.

An *inheritance concept lattice* (*ICL*) corresponds to the set of pairs (AN, BN). Figure 2.2.1 is the ICL for the example relation of Figure 2.1.1. From a class design perspective, the idea is to consider each concept of the generated *ICL* as a class and the edges of the Hasse diagram as the class-subclass relations. The *ICL* also shows where each property should be declared. One important property of the *ICL* is that each property appears exactly in one place. We obtain in such a manner a maximal factorization of the common properties of the classes. In the example, only message selectors are considered and therefore the hierarchy does not show where the different implementations of the same selector should be defined. This will be addressed in section 2.4.

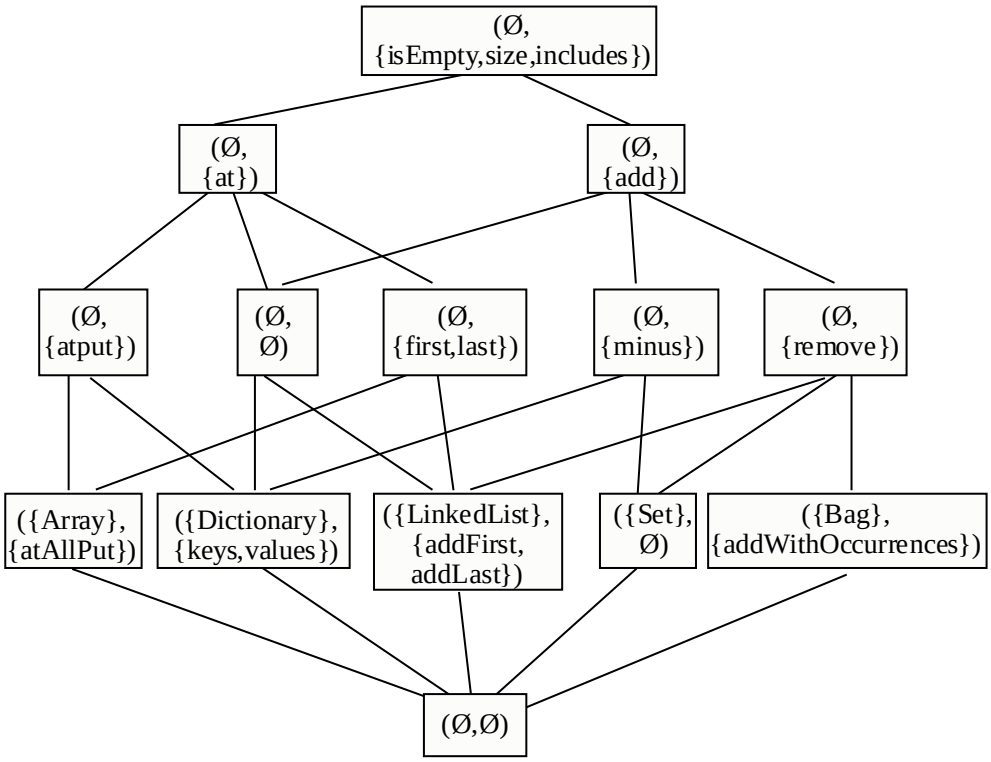

Figure 2.2.1. Inheritance concept lattice.

The representation used in (Cook, 1992) is based on the inheritance concept lattice. It shows explicitly the message selectors that are not inherited from the parent nodes, and thus that distinguish the set of classes represented by that node. Those message selectors can also be used to determine the semantics of the concept. For example, the concept (Ø,{at}) can be interpreted as the set of Indexed Collections responding to the at message which returns the element at a given index in the collection. In this manner each concept having an empty *AN* set can be considered as a new abstract class and names can be given to identify these. The names given in (Cook, 1992) for the example appear in Figure 2.2.2. Two abstract classes, Collection and SequenceableCollection, appear in the ObjectWorks Smalltalk library itself. There are however many other interesting abstractions which are revealed by the lattice. There still is one subtle difference between this structure and the hierarchy used in (Cook, 1992) because of the Indexed Extensible Collection class that does not appear in his structure. The difference will be clarified below.

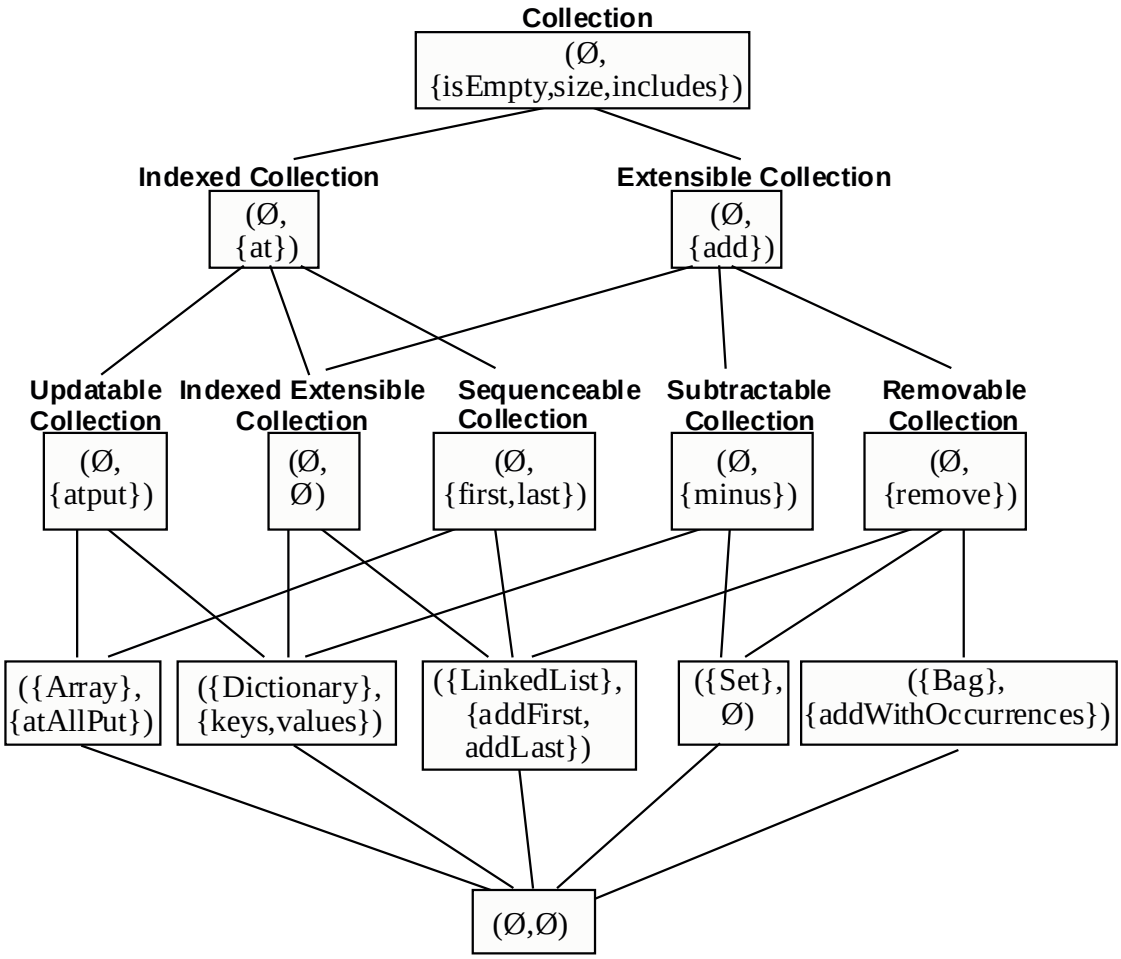

Figure 2.2.2. Inheritance concept lattice plus abstract protocol names.

The algorithm used for generating the concept lattice is described in (Godin et al., 1995d) and generates for each pair the sets *A*, *B*, *AN* and *BN*. In our tool, users can choose the representation that is the most useful for their purposes. When generating the class hierarchies, the *ICL* is used.

2.3 Pruned concept hierarchy

For some applications, the concept lattice contains too many concepts and a well chosen subset might be more useful (Godin, Mineau, Missaoui & Mili, 1995c). One interesting subset is the *pruned concept (galois) hierarchy* (also called *Galois subhierarchy* in (Dicky et al., 1994)) which has been introduced in the more general context of conceptual clustering of conceptual graphs, under the name *knowledge space* (Mineau, Gecsei & Godin, 1990). We describe the pruned concept hierarchy for the case of binary relations below.

The *pruned concept hierarchy (PCH)* may be generated from the concept lattice by eliminating the pairs which have *empty AN and BN sets*. In the example of Figure 2.2.2, two pairs would be eliminated, and the resulting *pruned inheritance concept hierarchy (PICH)* is shown in Figure 2.3.1. Although the maximal factorization is preserved in the pruned hierarchy, the lattice property is not always preserved. The interface hierarchy used in (Cook, 1992) corresponds to the *PICH*. The Indexed Extensible Collection node (class) was removed because all of its selectors came from parent classes. Whether this is a sufficient condition not to consider it is debatable. Under some circumstances, there are some good reasons to keep it in order to minimize multiple inheritance, for example. As an extreme case, consider a library where 4 classes all inherit from the same 4 classes in the *PICH* (Figure 2.3.2). This would be represented by 4 X 4 = 16 inheritance links. It would be preferable to add an intermediary class as in the concept lattice which would represent the common parts inherited from the 4 classes. The number of links would therefore be reduced to 4+4. A comparison using metrics for evaluating class hierarchies will be presented in the next section. Maintaining a lattice structure also has some advantages from an implementation point of view (Caseau, 1993).

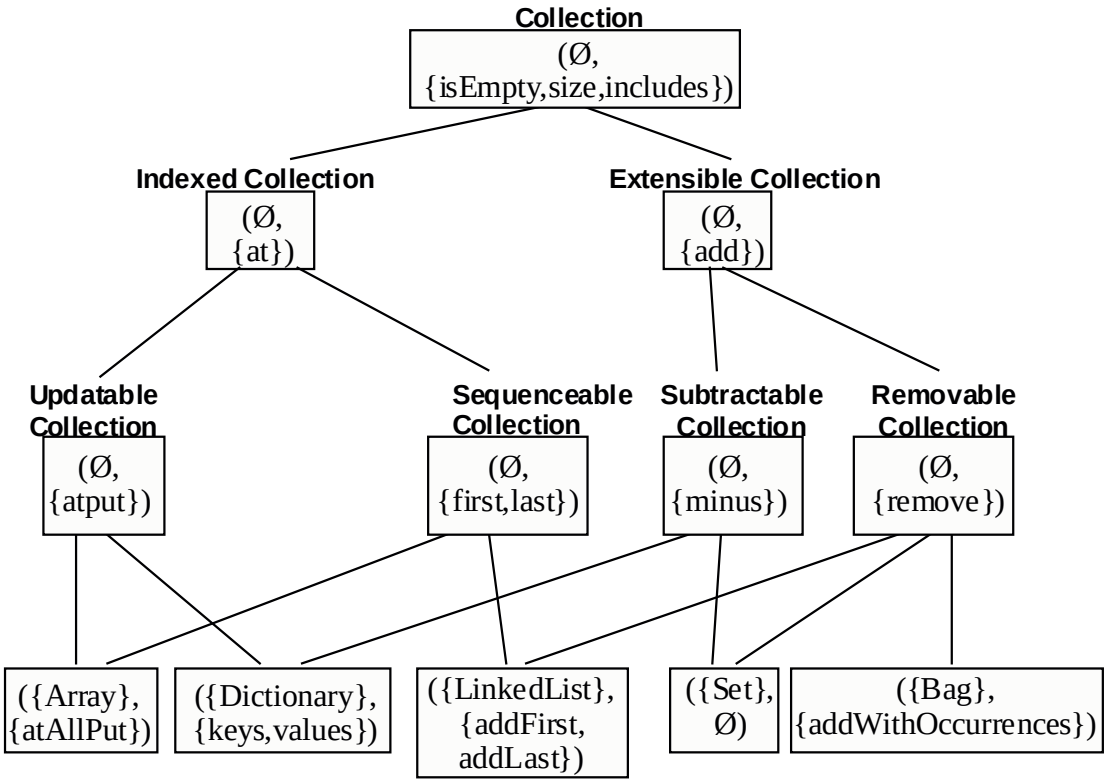

Figure 2.3.1. Pruned inheritance concept hierarchy (*PICH*) for the relation in Figure 2.1.1.

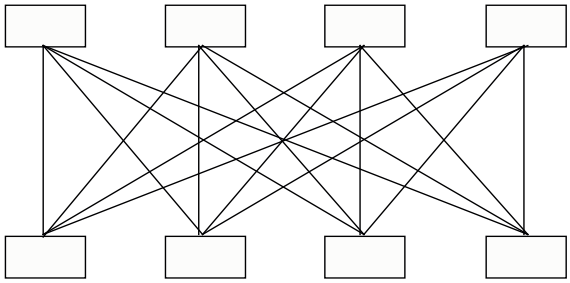

Figure 2.3.2. Example of 4 classes inheriting from 4 others in *PICH*.

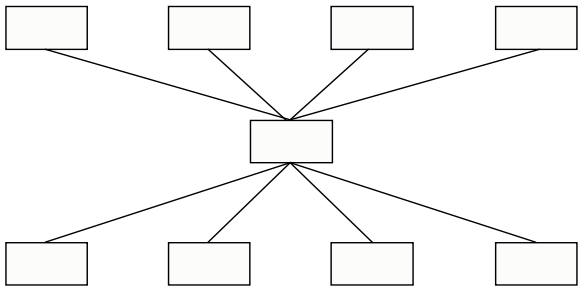

Figure 2.3.3. Same example with *CL*.

Rather than generating the *PICH* from the concept lattice by pruning empty concepts, incremental algorithms have been designed that generate the *PICH* directly. The ARES algorithm described in (Dicky et al., 1994) can generate the *PICH*. In our case, we have adapted the incremental algorithm for *ICLs* (Godin, Mineau & Missaoui, 1995b). Experiments described in Section 3 show that generating the *PICH* can be much more efficient than generating the *CL*.

2.4 Using taxonomic relations

The description of a class used in the previous sections consisted simply of a set of unrelated properties. The fact that these items corresponded to method selectors has no bearing on the technique. We could use attributes instead, or both, or any other specification in the form of a set of properties as in (Dvorak, 1994). However, the structures do not consider relationships between the properties. For example, it is important to be able to handle overloading where the same message selectors may have different implementations. We cannot consider such methods as completely different, for otherwise we miss a useful abstraction in the fact that they implement the same abstract behavior. We would like to have a hierarchy where both the commonalities of abstract interfaces and implementations are identified. It is also important to take into account the possibility that a method might be a specialization of another method and use this information in the classification process.

Using the framework described earlier, this can be accomplished by defining taxonomic relationships between class properties, which are then taken into account in the process of generating and comparing the concepts .As a first step, we only consider relations between the abstract selectors and the different implementations. This is the approach taken for our experiments. The result will show where each interface and implementation should be defined in the hierarchy in order to avoid duplication. In Smalltalk, the language does not provide a mechanism for defining the interface hierarchy. However, a commonly used practice in the Smalltalk library indicates that a method is abstract/deferred by having it generate an exception when executed, which in effect says that the method should be redefined in the subclasses and not called as is.

To apply the previous framework, we consider each implementation of the same selector as a specialization of the abstract behavior represented by the selector name. Figure 2.4.1 represents the relationship between classes and the implementations for our collection classes example. The different implementations for the message selector x are denoted by x_1, x_2, etc. For example, Set, Bag, Dictionary and Array all use the same implementation isEmpty_1 for the isEmpty method selector by

inheriting it from the Collection class, while LinkedList uses another implementation $isEmpty_2$. The relationship of Figures 2.1.1 and 2.4.1 with the actual library can be understood by examining Figure 2.4.2. Figure 2.4.2 shows where each method is defined in the actual ObjectWorks Smalltalk hierarchy. The right-hand column shows the methods defined in the class and the left-hand column represents the effect of the definitions on the actual interface of the class.

Note that for the case of Smalltalk, the interface of a class cannot be equated with the set of messages that the class understands. For instance, classes sometimes understand (inherit) methods that they should not call. There are two important cases where one does not want a class to call an inherited method. The first case is when a general-purpose inherited method is unsafe or otherwise inappropriate for the class at hand. Because there is no built-in mechanism for « canceling » an inherited method, a common practice used in the class library is to redefine such a « forbidden » method locally to raise an exception when invoked. This is denoted in Figure 2.4.2 by following the name of the method with a "/" and a code identifying the type of exception. For example, the atPut method is canceled in class LinkedList by raising an exception because we cannot access linked lists with an index. The second case where we would not want objects of a class to call an inherited method is when the intent of the inherited method was to define the selector as part of the abstract interface and to leave the actual implementation as a responsibility of the subclasses. In this case, it is the inherited method that raises an exception-- called SubResp, for *subclass responsibility*. In Figure 2.4.2, new selectors in the interface of a class with respect to the parent class appear in bold on the left-hand column and cancelled selectors appear in gray and italic.

To represent the relationship between the message selector and the implementations, a taxonomic relation, noted < is defined:

$isEmpty_1 < isEmpty$,

$isEmpty_2 < isEmpty$, etc.

The taxonomic relation should not be confused with the partial order between concepts in the hierarchy. The definitions in the previous sections are modified simply by extending the relation I in order to take into account the taxonomic relation. We define a new relation I^+ between elements of *G* and *M* that includes the method selectors and the representation of the implementations:

gI^+m if $\exists m'$ such that $m' \leq m$ and $g\mathrm{I}m'$.

Set *M* now contains both the message selectors and methods. In our example, Set I^+ $\mathsf{isEmpty}_1$ and in addition, Set I^+ isEmpty. Replacing *I* with I^+, we obtain the enriched context (*G*, *M*, I^+) where the taxonomic relationships are taken into account.

In our example, the pair ({Dictionary, LinkedList}, {isEmpty, size, includes, $\mathrm{includes}_1$, add, at}) is a concept for *CL*(*G*, *M*, I^+). Although the implementations are different for isEmpty, size, add and at, they match on the selector name using the taxonomic relation. The resulting lattice appears in Figure 2.4.3. Compared to the previous lattice where only the interface appeared, there are new concepts that are highlighted in bold squares. For example, the new concept ({Array, Dictionary, Bag, Set}, {isEmpty, $\mathrm{isEmpty}_1$, size, includes}) emerges. It represents the set of classes that respond to the common interface {isEmpty, size, includes} and in addition share the same implementation $\mathsf{isEmpty}_1$. To distinguish this type of hierarchy from interface hierarchies, we call it an *Int-Imp hierarchy*. As before, the notions of inheritance and pruning can be applied. The *ICL* appears in Figure 2.4.4. Again, this structure is the most relevant one for class hierarchies because it shows where the interface and implementations should be defined for each selector. The result is a hierarchy where there is no redundancy for the implementation and interface definitions and which is consistent with specialization. Furthermore, the fact that the methods are considered as completely different implies that methods are never redefined but only deferred methods for which the interface have been declared can be defined. The *PICH* is shown in Figure 2.4.5. Compared to the *PICH* based only on the interface, there is one additional class (in a bold square) for the set of classes sharing the $\mathsf{isEmpty}_1$ implementation.

It would evidently be possible to generate the hierarchies based only on the implementation specifications without using the taxonomic relation with the interfaces. We call this an *Imp* hierarchy. This would show

how to build a hierarchy where there is no code duplication. As opposed to the *Int-Imp* hierarchy, there would be some interface redundancy, however, because the common interfaces are not detected unless they all share the same implementation. Nevertheless, there might be circumstances where this is an interesting compromise.

So far, we have presented the case where taxonomic (conformance) relations existed only between signatures and implementations. Our experiments are based on this type of specification. However, this is not an inherent limitation of the approach. The preceding taxonomic relation is limited by the fact that the implementations are considered as completely different from each other. There might be implementations that are specializations of other implementations and should not be considered as completely different. Redefining a method by specializing it should be permitted and would produce hierarchies which are also consistent with specialization. This can be taken into account by extending the taxonomy to incorporate these specialization relationships between method implementations. This results into an even better hierarchy. We have not used this ideal approach in our experiments because we could not infer these relationships from an automatic code analysis. However, if such information were available it could be incorporated in the taxonomy. The taxonomic relationships between methods could be inferred from formal specifications or supplied by the designer. Dicky et al. Showed how some special cases can be computed (Dicky et al., 1996). The result will be a hierarchy where method redefinition is permitted when consistent with specialization and that is *maximally factorized* as defined in (Dicky et al., 1996) ; an inheritance hierarchy is maximally factorized, if and only if, whenever $Class_1$ has property x_1 and $Class_2$ has property x_2, and x_3 is the least upper bound of x_1 and x_2, then the hierarchy has a common superclass of $Class_1$ and $Class_2$ containing the declaration of x_3.

Consider the example in Figure 2.4.6. $Class_1$ uses methods $\{a_1, b_1\}$, $Class_2$ uses $\{a_1, b_2\}$, and $Class_3$ uses $\{a_2, b_1\}$. The taxonomy for these methods is given in Figure 2.4.7. As shown, in addition to the relationship between the methods and their interfaces, Method a_2 is a specialization of a_1 and b_2 is a specialization of b_1. If we only take the relationship between methods and interfaces, we obtain the *ICL* in Figure 2.4.8. When taking the specialization relationships between methods into account, the *ICL*

produces a much more compact and tight hierarchy as illustrated in Figure 2.4.9. Failing to recognize the specialization relationships, many additionnal concepts have to be generated resulting in an unnecessarily complicated design. For example, in Figure 2.4.8, the *ICL* fails to recognize that $Class_2$ is a specialization of $Class_1$ because it considers b_1 and b_2 to be unrelated except for their common interface. It therefore creates a new concept to factor out specifically the a_1 method that is common to $Class_1$ and $Class_2$. The result is not maximally factorized because there is no common superclass of $Class_1$ and $Class_2$ that declares b_1, the least upper bound of b_1 and b_2. When using the taxonomy as in Figure 2.4.9, the *ICL* is maximally factorized because the common superclass of $Class_1$ and $Class_2$, which is$Class_1$, declares b_1, the least upper bound of b_1 and b_2. The same phemenon can be observed for $Class_1$ and $Class_3$ with respect to a_1 and a_2. When using PICH, the same advantages are produced. In fact, except for the bottom concept, the PICH is equal to the ICL for the previous example.

The approach presented so far uses only the methods for classification but it can be applied to any type of specification of classes in the form of sets of properties which may be related by a partial order. For example, Dvorak (Dvorak, 1994) proposed a classification algorithm based on formal specifications of conceptual class properties. Our approach is directly applicable on that type of representation.

Finally, the approach can been generalized to handle richer descriptions in the form of conceptual graphs (Mineau & Godin, 1995). It can therefore be applied to handle richer descriptions of the classes (Godin & Mili, 1993). The current interest in formal specifications for classes (Cheon & Leavens, 1994; Parisi-Presicce & Pierantonio, 1994) suggests that these might be available on a larger scale in the future and could be exploited by classification algorithms.

class → method ↓	Set	Bag	Dictionary	Linked List	Array
$isEmpty_1$	1	1	1	0	1
$isEmpty_2$	0	0	0	1	0
$size_1$	0	0	0	1	0
$size_2$	1	0	1	0	0
$size_3$	0	1	0	0	0
$size_4$	0	0	0	0	1
$includes_1$	0	0	1	1	1
$includes_2$	1	0	0	0	0
$includes_3$	0	1	0	0	0
add_1	1	0	0	0	0
add_2	0	1	0	0	0
add_3	0	0	1	0	0
add_4	0	0	0	1	0
$remove_1$	1	0	0	0	0
$remove_2$	0	1	0	0	0
$remove_3$	0	0	0	1	0
$minus_1$	1	0	1	0	0
$addWithOccurren_1$	0	1	0	0	0
at_1	0	0	0	0	1
at_2	0	0	1	0	0
at_3	0	0	0	1	0
$atput_1$	0	0	1	0	0
$atput_2$	0	0	0	0	1
atAllPut1	0	0	0	0	1
$first_1$	0	0	0	0	1
$first_2$	0	0	0	1	0
$last_1$	0	0	0	0	1
$last_2$	0	0	0	1	0
$addFirst_1$	0	0	0	1	0
$addLast_1$	0	0	0	1	0
$keys_1$	0	0	1	0	0
$values_1$	0	0	1	0	0

Figure 2.4.1. Relation I between classes and implementations.

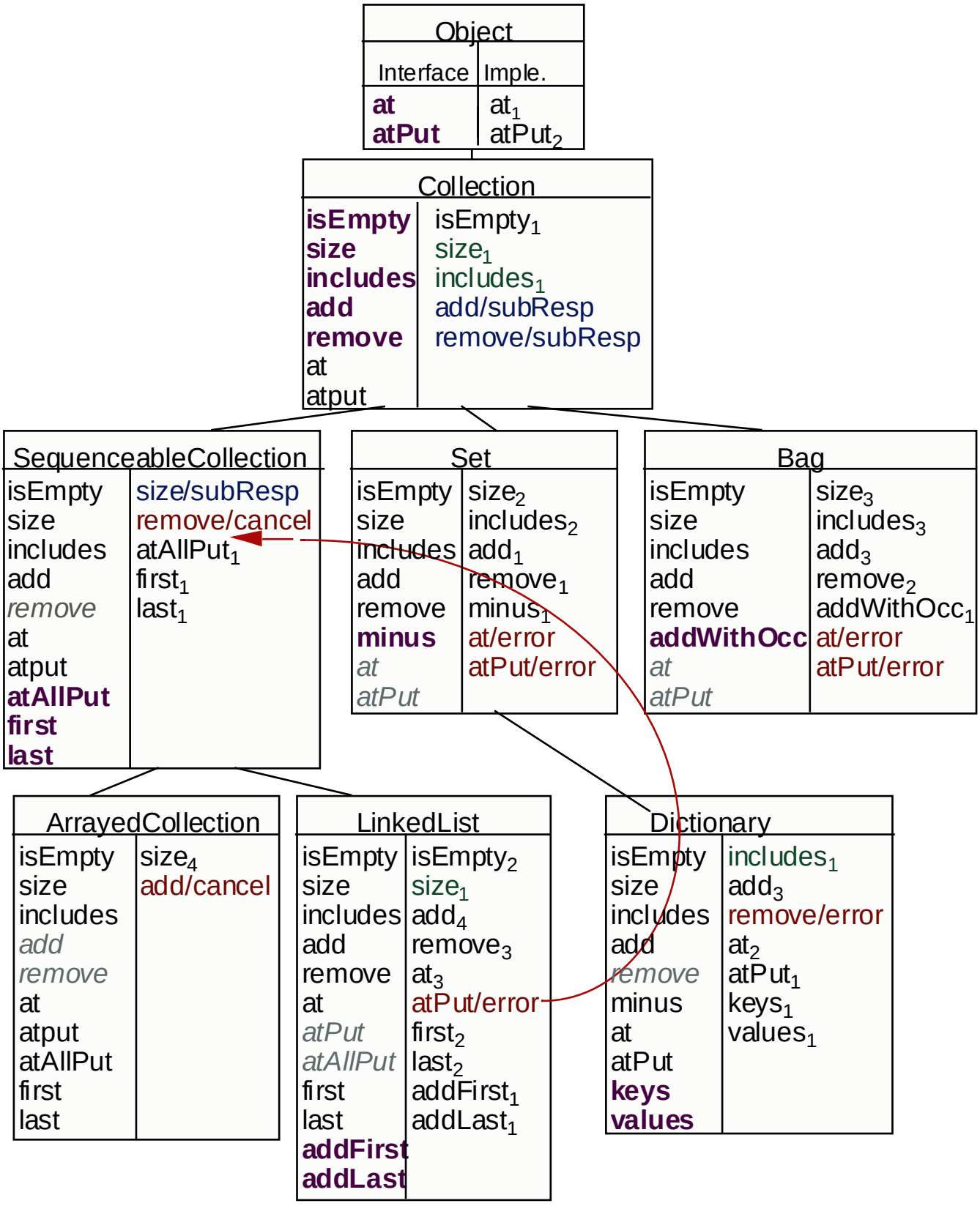

Figure 2.4.2. Subset of the ObjectWorks Smalltalk class hierarchy.

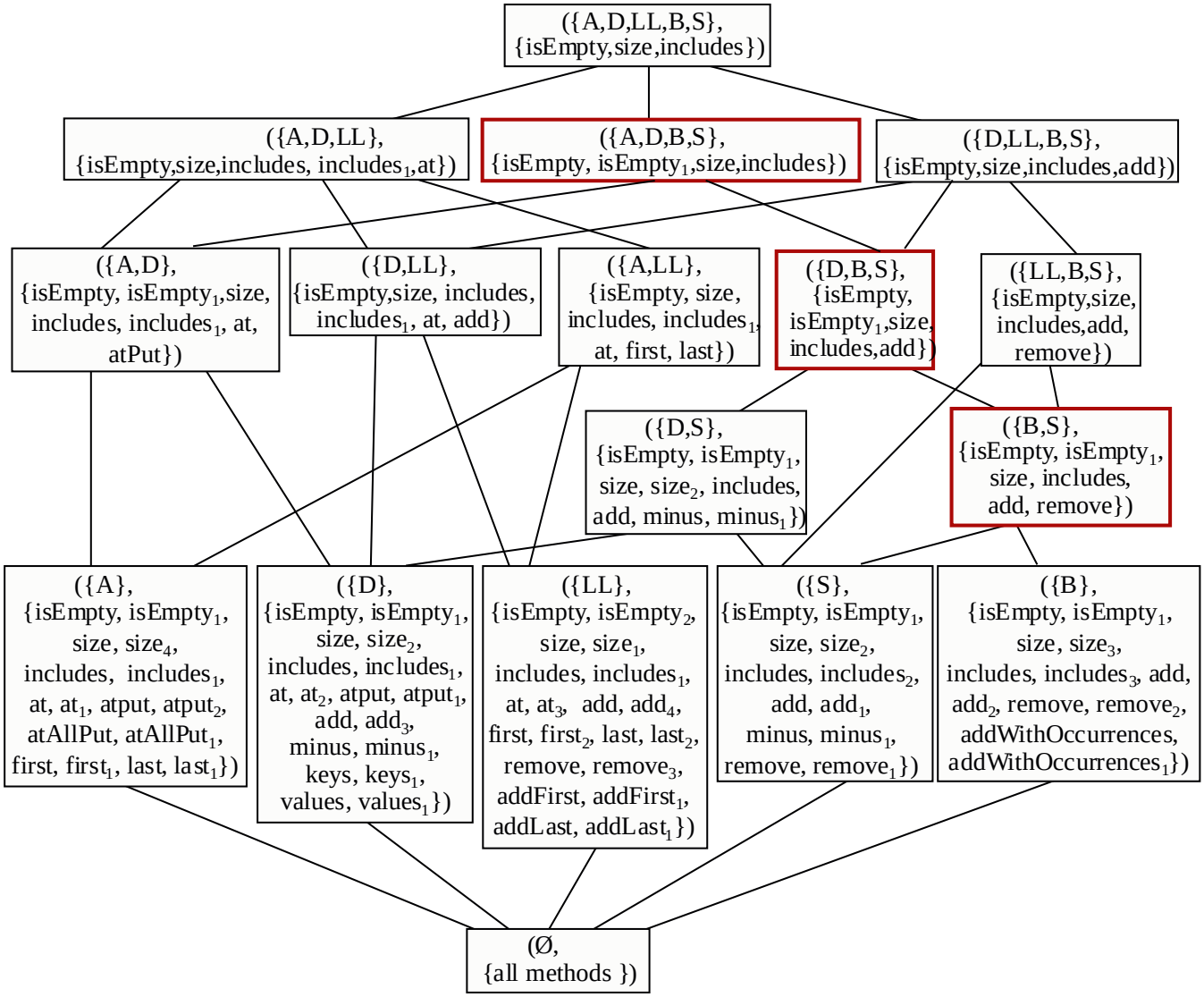

Figure 2.4.3. Concept lattice *CL* for interface and implementation (*Int-Imp CL*).

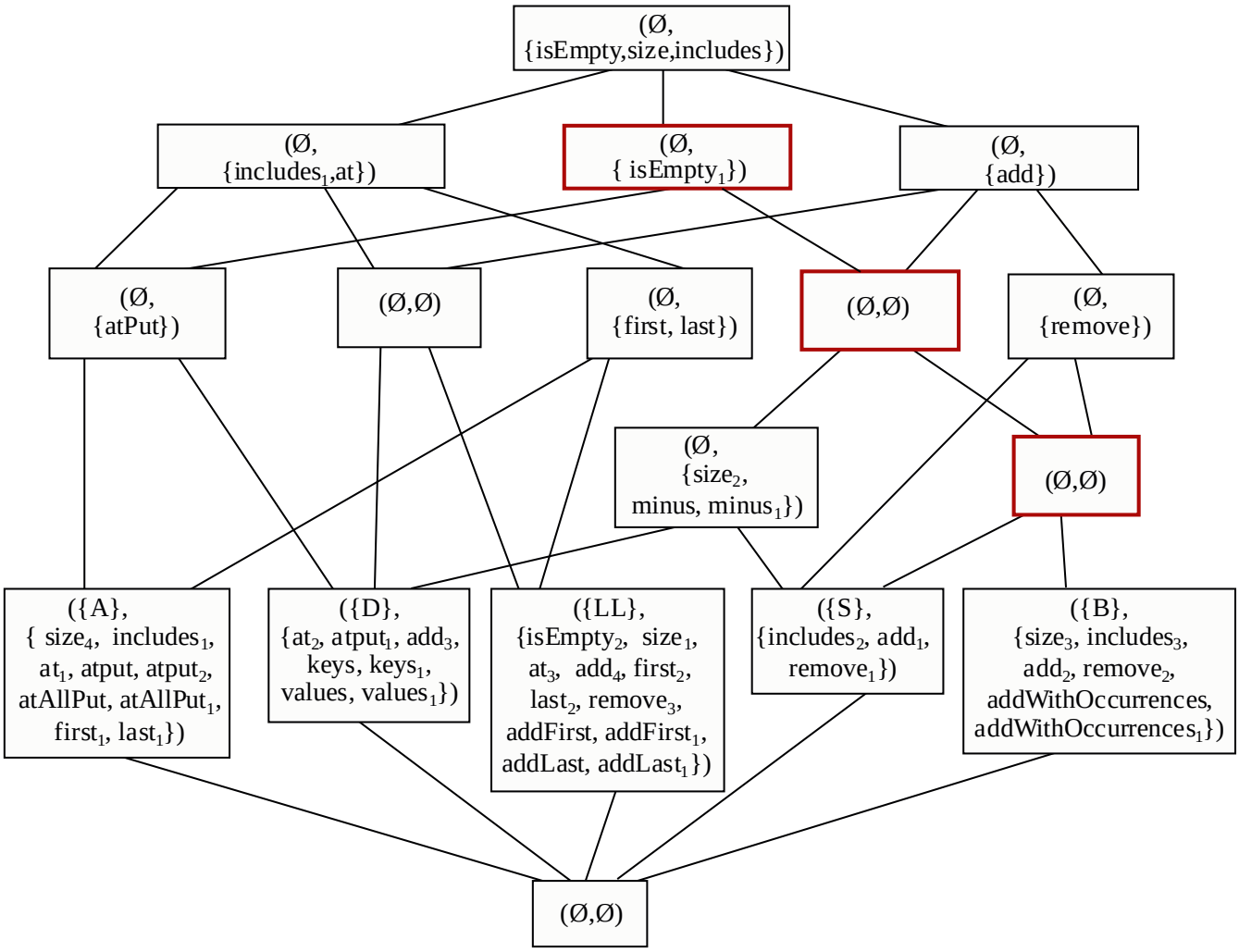

Figure 2.4.4. *ICL* for interface and implementation (*Int-Imp ICL*).

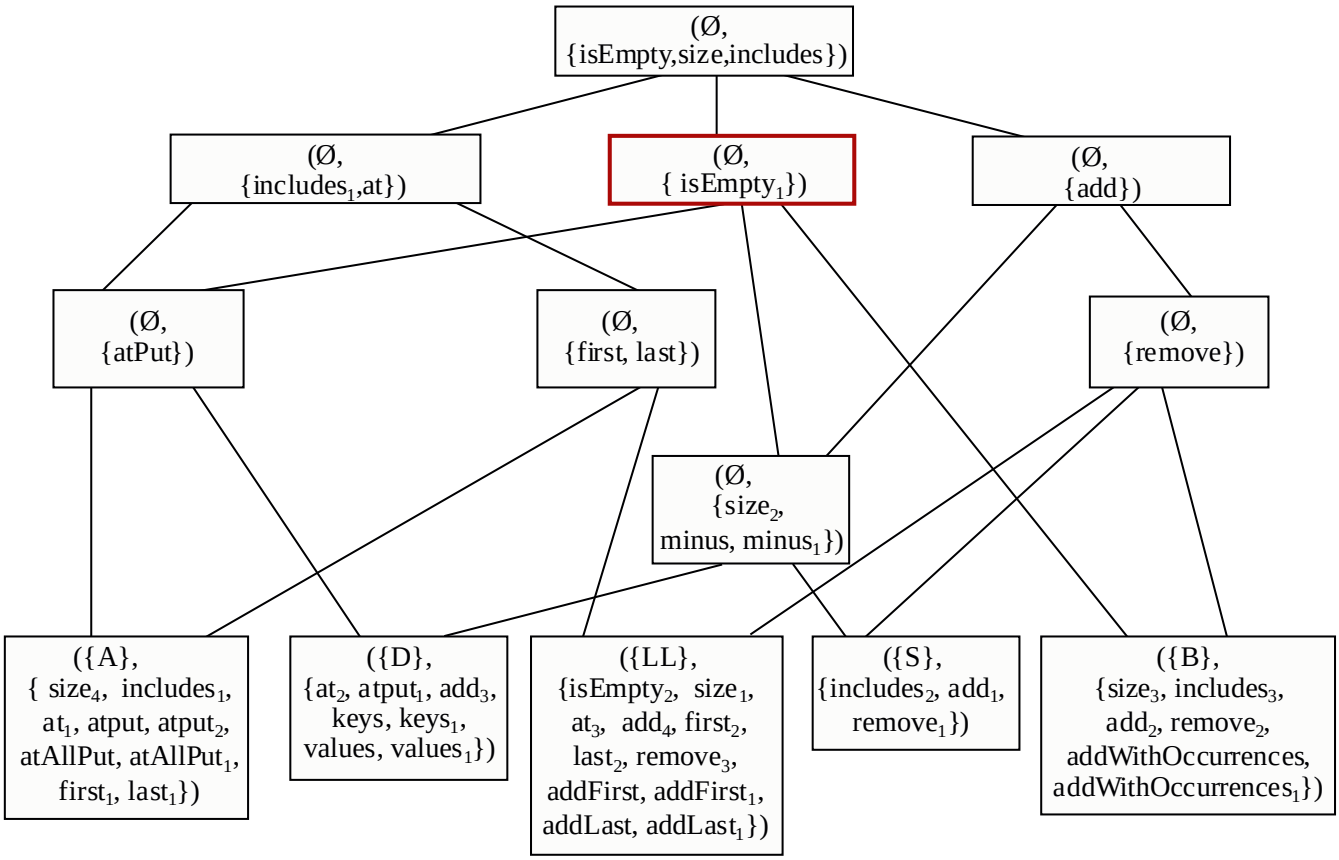

Figure 2.4.5. Pruned inheritance concept hierarchy (*PICH*) for interface and implementation (*Int-Imp PICH*).

class → method ↓	$Class_1$	$Class_2$	$Class_3$
a_1	1	1	0
a_2	0	0	1
b_1	1	0	1
b_2	0	1	0

Figure 2.4.6. Relation between classes and implementations.

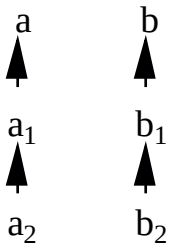

Figure 2.4.7. Specialization relationship between methods.

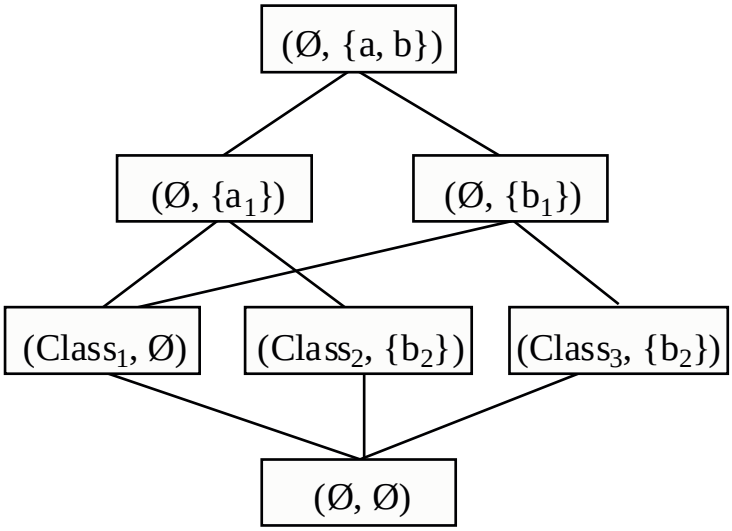

Figure 2.4.8. *ICL* without using specialization relationship between methods.

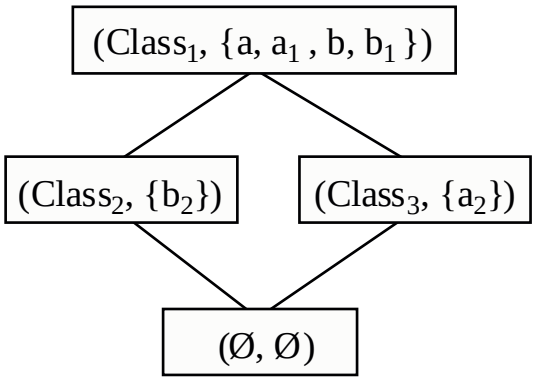

Figure 2.4.9. *ICL* using specialization relationship between methods.

3. Structural metrics

The metrics we are seeking should help to compare different class hierarchies as a whole. They do not evaluate the quality of the initial classes themselves used as a starting point for building the hierarchies. There might be alternative designs of these initial classes which may involve splitting or combining classes ; this is another important problem which is the focus of some ongoing work..

Chidamber and Kemerer (1994) have proposed a set of six metrics which apply to classes as individuals. Their metrics are useful to compare different designs of the individual classes. Nevertheless, some of these local metrics are related to the global metrics used here for the hierarchy as a whole. To our knowledge there are no empirically validated metrics for the design of class hierarchies. However, based on the current work (Casais, 1991; Johnson & Foote, 1988; Korson & McGregor, 1992; Lieberherr et al., 1991), there are at least three aspects which seem important in determining the quality of class hierarchies and should therefore be measured: redundancy, complexity and deviation from specialization.

The work of the Northeastern University group on the Demeter System has addressed the problem of a quality measure for class hierarchies in their approach to automatic discovery of class hierarchies from example objects (Lieberherr et al., 1991). Their algorithm uses a metric which underlies the optimization process. They use a two step learning algorithm where the first step does basic learning by generating a potentially non-optimal class dictionary graph. The edges of the graph represent the inheritance (alternation edges) and part-of (construction edges) relationships between classes. The second step optimizes the graph by trying to minimize a weighted function of the edges where the weight of the construction edges is four times the weight of the alternation edges (they suggest that any value larger than two would be acceptable). This function can be considered as a metric for the class hierarchy. In their work, a part is considered as a high-level concept which might be implemented as a method or as an instance variable. In our work, we use the methods as parts. The total number of parts cumulated by adding the number of parts for each class will be called *Parts* in the following. The individual metric computed for each class is similar to the *Weighted Methods Per Class* metric proposed in (Chidamber &

Kemerer, 1994). However the perspective is different in this context because it is not used to evaluate the complexity of individual classes. The total number of inheritance edges in the inheritance graph will be denoted *Inher*. This corresponds to the *Number of Children* metric of (Chidamber & Kemerer, 1994) which is summed up for all classes. The metric of (Lieberherr et al., 1991) is called *M1* in the following and is defined as:

$$M1 = 4*Parts + Inher.$$

Using this function favors factoring out common parts which reduces the construction edges at the expense of adding alternation edges. This is consistent with the idea of minimizing redundancy. It is a good principle from a software engineering point of view since having each thing in one place facilitates comprehension and maintenance (resiliency to change (Lieberherr et al., 1991)). Overall, the function also tries to minimize the complexity of the hierarchy measured by counting the edges.

The above metric has some weaknesses. For example, the two designs in Figures 3.1-a and 3.1-b corresponding to the *PICH* and *ICL* would produce the same value for the above metric and Lieberherr et al.'s algorithm (Lieberherr et al., 1991) would not be able to choose between the two. In the first case, there is more multiple inheritance since both bottom classes inherit parts *a* and *b* from two other classes. It seems preferable in general to favor the second case where the common parts are put together and single inheritance is used for the two bottom classes. Given that the second design seems preferable, we could therefore try to minimize multiple inheritance by adding some measure of this to the metric *M1*. Using the amount of multiple inheritance was proposed as future research in (Lieberherr et al., 1991) . Although multiple inheritance is useful in some cases, it should only be used when necessary. All other things being equal, it seems reasonable to favor designs using less multiple inheritance. At the extreme case, if single inheritance is sufficient, it should be favored. To measure the amount of multiple inheritance, we can count the number of parents minus one for each class whose number of parents is greater than one. This measure will be called *Mult* in the following. This amounts to giving double weight to parent edges except for the first one.

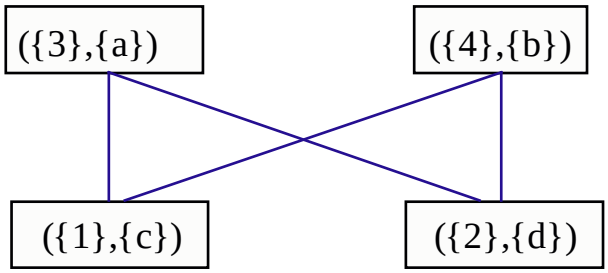

Figure 3.1-a. Design 1.

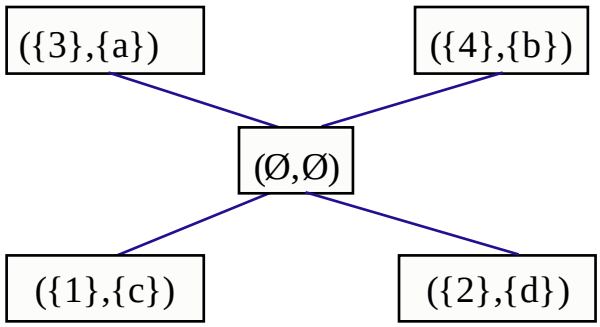

Figure 3.1-b. Design 2.

Another weakness of metric *M1* is that it does not measure deviation from specialization. This is correct in the context of the algorithm used in (Lieberherr et al., 1991) because it always produces hierarchies which are consistent with specialization. Many have argued that the hierarchy should be consistent with specialization as much as possible to enhance understandability (Casais, 1991; Coleman et al., 1994; Korson & McGregor, 1992; Liskov, 1988). A rough measure of deviation from specialization could be produced by computing the difference between the union of methods of the class and its ancestors and the computed interface of the class. This reflects the cancellations and dependencies on canceled methods which affect the actual interface of a class. The measure is summed for each class in the hierarchy and the total is called *Dev* in the following. For the concept lattice and variants, *Dev* is always zero. One important drawback of this measure is that it does not penalize redefinitions which are not specializations. Clearly, this could be taken into account if the taxonomic relations between methods were known. However, since we do not have this data for our experiments, we have chosen not to penalize redefinitions. This does introduce a bias in favor of Smalltalk because there are some cases when the redefinitions are not specializations.

Adding measures for the amount of multiple inheritance and deviation from specialization to metric *M1* gives metric *M2*:

$$M2 = M1 + Mult + Dev.$$

Adding these together is somewhat arbitrary and the relative weights of each part should depend on the goals of the design. Our tool does give the individual values for each aspect.

4. Implementation and experiments

4.1 Overview of the tool set

A tool for producing the concept lattice and the variants described in the previous section was developed using ObjectWorks Smalltalk. The input, i.e. names of the classes and their properties, can come from external sources or can be extracted from the Smalltalk library itself. The properties can be ordered using a taxonomy. Many algorithms are implemented for building the various structures. All the algorithms are incremental. The algorithms can be used to add a new class, delete an existing class or update the specifications of a class and reflect the modifications on the hierarchies. If the designer wants to build a hierarchy from a batch of classes, we use the incremental algorithms to build the hierarchy incrementally by integrating the classes one by one. The resulting hierarchies can be examined and interactively refined using a graphical browser. The final result could be used to generate the code for the class hierarchy whereby each (Galois lattice) concept is mapped to a class declaration in a target language.

A number of incremental algorithms based on (Godin et al., 1995d) have been implemented to update the *CL*. From the *CL*, the *PICH* can be generated. The algorithm for doing so is straightforward: it goes through each concept one by one and eliminates the unwanted concepts. We have also implemented an incremental algorithm based on (Godin et al., 1995b) to create and update the *PICH* directly without generating the *CL*. As will be shown later, the *PICH* algorithm can be much more efficient than the *CL* algorithm in some cases. These algorithms have also been extended to handle taxonomic relationships between the properties as described in section 2.4.

Despite the flexibility and the performance of the above algorithms, it would be unrealistic to expect that the hierarchies produced be used as is. Instead, they should be considered as a starting point where all the potential commonalities have been extracted. However, the designer might consider eliminating such commonalities or interactively delete concepts from the structures based on his/her subjective evaluation; the tool will automatically modify the hierarchy so that nothing is lost in the process. If the starting point is the *CL*, the empty classes are candidates for deletion. Empty classes are extreme cases, but near empty classes could also be considered.

The tool also collects several measures related to the quality of the produced hierarchy. We have used these metrics to compare the hierarchies produced by our algorithms and sub-hierarchies of the Smalltalk library. The tool can compute the metrics for any subtree of the Smalltalk library taking into account an increasing number of classes by adding each class one by one using a depth first search of the hierarchy. In our experiment, the same classes are used in the same order to incrementally build the *ICL* and *PICH*.

4.2 Building the interface hierarchy for the Smalltalk `Collection` library

4.2.1 Extracting class interfaces

The classification algorithms were applied to specifications of the classes extracted from the ObjectWorks\Smalltalk code. To extract such specifications, we used an approach similar to that taken by Cook (Cook, 1992). Extracting the interface (protocol) of a given class is more involved than simply computing the union of all selectors for methods defined by the class and its superclasses. For example, a subclass method may cancel a method inherited from its superclasses by redefining it to return an error. Many examples appear in Figure 2.4.2. For example the atPut method is canceled in class LinkedList by returning an error. The code of the canceling method simply consists of sending a special error method to the receiver of the message (self). This can be automatically detected by a simple syntactical analysis of the code of the methods. Our tool starts out with the union of all selectors and then examines the corresponding method implementations to determine if they canceled inherited methods by providing an error notification ; such methods are eliminated from the interface of the class.

Another problem to resolve is typical of the conventional use of abstract classes. These classes include methods that return a typical error indicating they should be implemented by subclasses. However, other methods depend upon these methods. Thus, if the subclass does not implement a given method, then all methods depending on it must be removed from the interface. For example, the atAllPut method defined in the SequenceableCollection class uses the atPut method. By canceling the atPut method in the LinkedList class, the atAllPut cannot be executed. The arrow in Figure 2.4.2 illustrates this dependency. Figure 2.4.2 shows the interface and methods implemented for each class. The left-hand column lists the extracted interface specification. Selectors which are not inherited but defined in the class appear in bold. Canceled selectors appear in gray and italic. The right-hand column shows which method is defined in the class. Cancellations are indicated by following the name of the method with a slash and a reference to the name of the error raised by the method. Dependencies on unimplemented methods occur very often in the Smalltalk classes, and are a source of confusion in trying to understand their behavior. The hierarchies produced with our algorithms are consistent with specialization and are thus guaranteed to avoid any cancellation.

The previous approach to computing the interface evidently relies on the fact that consistent method cancellation conventions are used throughout the library, which seems to be largely true. There are some difficult cases however, where the error messages sent depend on the class of the executing object. Some of these were treated as special cases in our tool. Other dependencies which would not follow these conventions are not taken into account. Therefore the extracted interface may in some cases be approximate.

Our prototype can produce specifications of the interface and/or implementations of the method selectors in the interface for any subset of classes which is a subtree of the inheritance hierarchy. From the set of class specifications extracted, the user can then incrementally generate class hierarchies using the algorithms. When the specification is limited to the interface for each class, the generated hierarchy corresponds to the interface hierarchy as in the examples based on Figure 2.1.1. The implementation

specifications are in the same format as the example in Figure 2.4.1. For each method *x* in the interface of the class, the tool gives an identifier for the implementation used in the format *xi* where *i* is a sequence number from 1 to n, n being the number of different implementations encountered. In addition, the taxonomy is generated relating each implementation *xi* to the abstract interface *x*. The user can then generate the hierarchies for the interface alone or for both the interface and implementation specifications by using the taxonomy. It is also possible to generate the hierarchies based on the implementations alone. The result would be useful to reorganize the hierarchy in order to obtain zero code redundancy. The result might contain some interface redundancy however. Depending on the priorities of the designer, this might be a valuable compromise because the number of classes is smaller. In our experiments, both possibilities are examined.

4.2.2 Time performance of the incremental algorithms

Figure 3.2 shows the CPU time for generating various structures using the incremental algorithms. The Figure contains data for the interface specifications alone (*ICL-Int*, *PICH-Int*), the implementation specifications alone (*ICL-Imp*, *PICH-Imp*) and the two used together (*PICH-Int+Imp*). The time for generating the *ICL-Int+Imp* does not appear because it is much larger and we would not be able to distinguish the others when combined on the same Figure. In this experiment, the *PICH* can be generated much more efficiently than the *ICL* for every combination. One interesting fact is that although the impact of using the taxonomic relationship is very high for the *ICL* (the time to add a class to the *ICL-Int+Imp* is about 50 times as large as the time to add a class to the *ICL-Imp*), it is relatively small for the *PICH* algorithm (about 10% difference).

4.2.3 Comparing hierarchies using metric values

Figure 3.3 shows metric *M2* for:

1) the sub-hierarchy of 64 classes rooted at the Collection class in the Smalltalk library (including a few classes of our own),
2) the inheritance concept lattice combining implementation and interface specifications using a taxonomic relationship (*ICL-Imp+Int*)

3) the pruned inheritance concept hierarchy combining implementation and interface specifications using a taxonomic relationship (*PICH-Imp+Int*)

Figure 3.3 shows the evolution of the metric as new classes are considered. This helps to reveal the trend in the growth pattern. As can be observed from the figure, the *PICH* shows the best results. The trend shows that the difference gets larger as the number of classes grows. The *ICL* is better than Smalltalk when the number of classes is limited. However, when the number of classes grows past 40, the *ICL* gets worse. The additional complexity of the empty classes seems to become prohibitive when the number of potential combinations of common properties gets large. This phenomenon may be especially true in the domain of Collection classes which are tightly connected to each other.

The good behavior of the *PICL* compared to the *ICL* also means that the actual occurrences of the pattern presented in Figures 2.3.2 and 2.3.3 are not very frequent. Most of the time the empty classes add to the complexity of the inheritance graph. This is the case for our example hierarchy in Figure 2.2.2. From the point of view of its children, Dictionary and LinkedList, The IndexedExtensibleCollection empty class is not necessary because all the method selectors inherited from IndexedExtensibleCollection by the children, namely {at, add}, are also inherited from others paths (repeated inheritance). Therefore, in the PICL, the empty class IndexedExtensibleCollection is simply removed with its inheritance edges without having to add any other inheritance edge. Given that in some cases the empty classes may be useful, we might want to consider an intermediary structure between the *ICL* and *PICH* which only keeps some empty classes based on some criteria. We plan to study this venue in future work.

We have made similar comparisons on other sub-hierarchies and observed similar patterns. However, these results should not be interpreted as a direct evaluation of the ObjectWorks\Smalltalk class hierarchy. One obvious point is that ObjectWorks\Smalltalk is limited to single inheritance which usually will result in designs having either redundancy or deviation from specialization or both. The Smalltalk library is used here as an example of the behavior of the algorithms on non trivial cases.

4.2.4 Observations

As previously mentioned, one important limitation of these experiments is that we distinguish between several implementations of the same selector simply by considering each method declaration as a different method unrelated to other methods. There are some cases where methods which are declared independently are in fact behaviorally equivalent or related by specialization. The quality of the produced hierarchies would be better if this information could be taken into account in the hierarchy building process. However, because of the difficulty of extracting this information from the code, we did not take it into consideration in our experiments. We have also explained how metric M2 should be enhanced by penalizing redefinitions that are not consistent with specialization. When information concerning how methods are related is available, it could be exploited by the algorithms. Typed object-oriented languages offer some better potential to exploit in this perspective and further experiments should be performed on such languages.

A related problem with the experiments is that the process relies on the quality of the naming conventions. It considers two methods with the same selector as being semantically related and conversely, different names are assumed to be unrelated. Here again the use of formal specifications could be used to detect these problems.

4.3 Organizing specifications from a (telecom. network) management information base

Another experiment was performed on a class hierarchy designed for a network management application within the IGLOO project. The objective was to produce an object model for Management Information Bases (MIB) using standard, publicly available specifications. Only the attributes were specified in the model. The initial manual design was implemented in Smalltalk and the algorithms were used to reengineer the class hierarchy. Several undetected commonalities in the initial design were revealed by the tool and were incorporated in the final design. Figure 3.3 shows metric *M2* for the initial design, the *ICL* and the *PICH*. Here again the *PICH* gives the best result. As opposed to the Smalltalk data, the behavior

of the *ICL* is also always better than the initial design. Because of the different configuration of the data, the number of empty classes does not grow as fast as for the previous experiment. The differences between the designs are not as large as for the Smalltalk data because the classes are not as tighly connected are there are less commonalities to uncover and factor out.

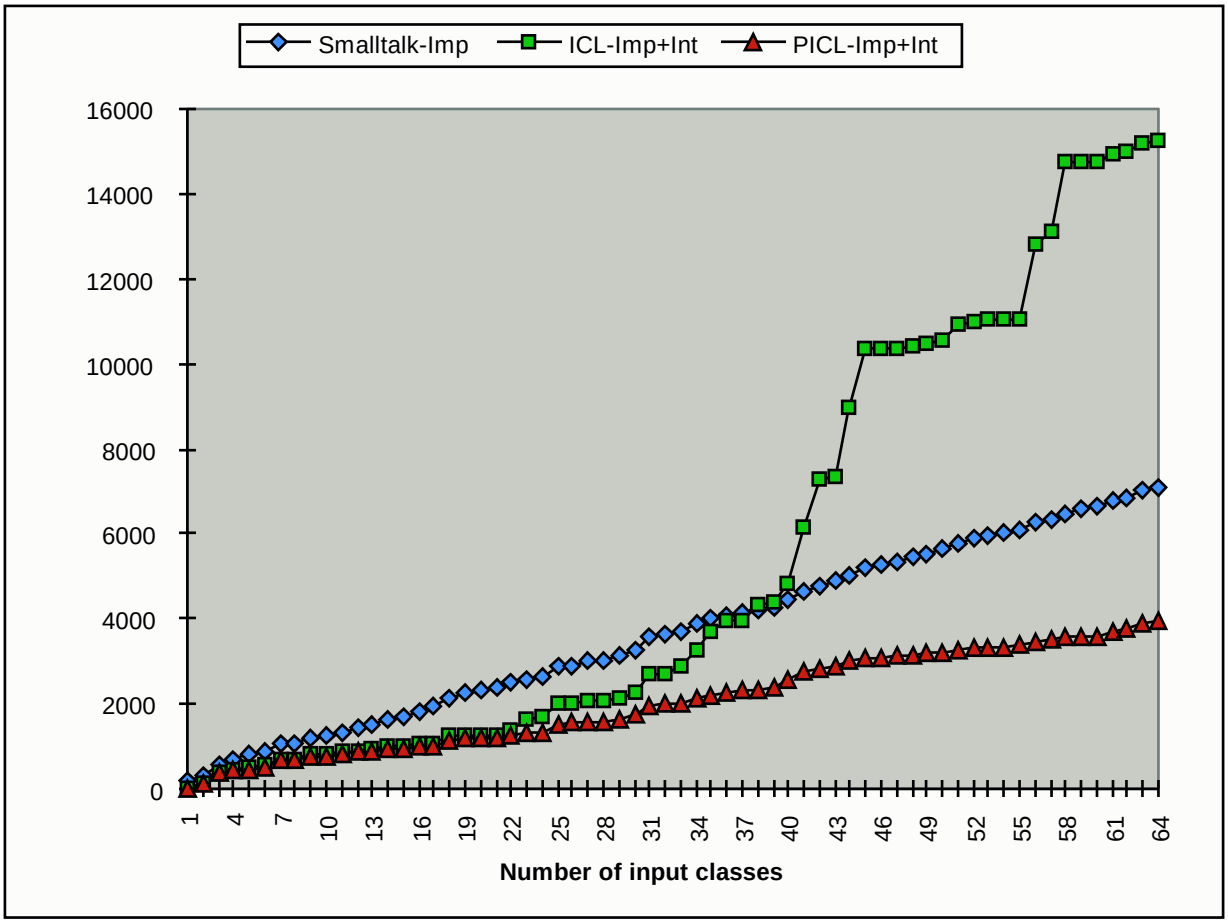

Figure 3.3. Metric *M2* for implementation and interface (*Int-Imp*) hierarchy for Collection classes.

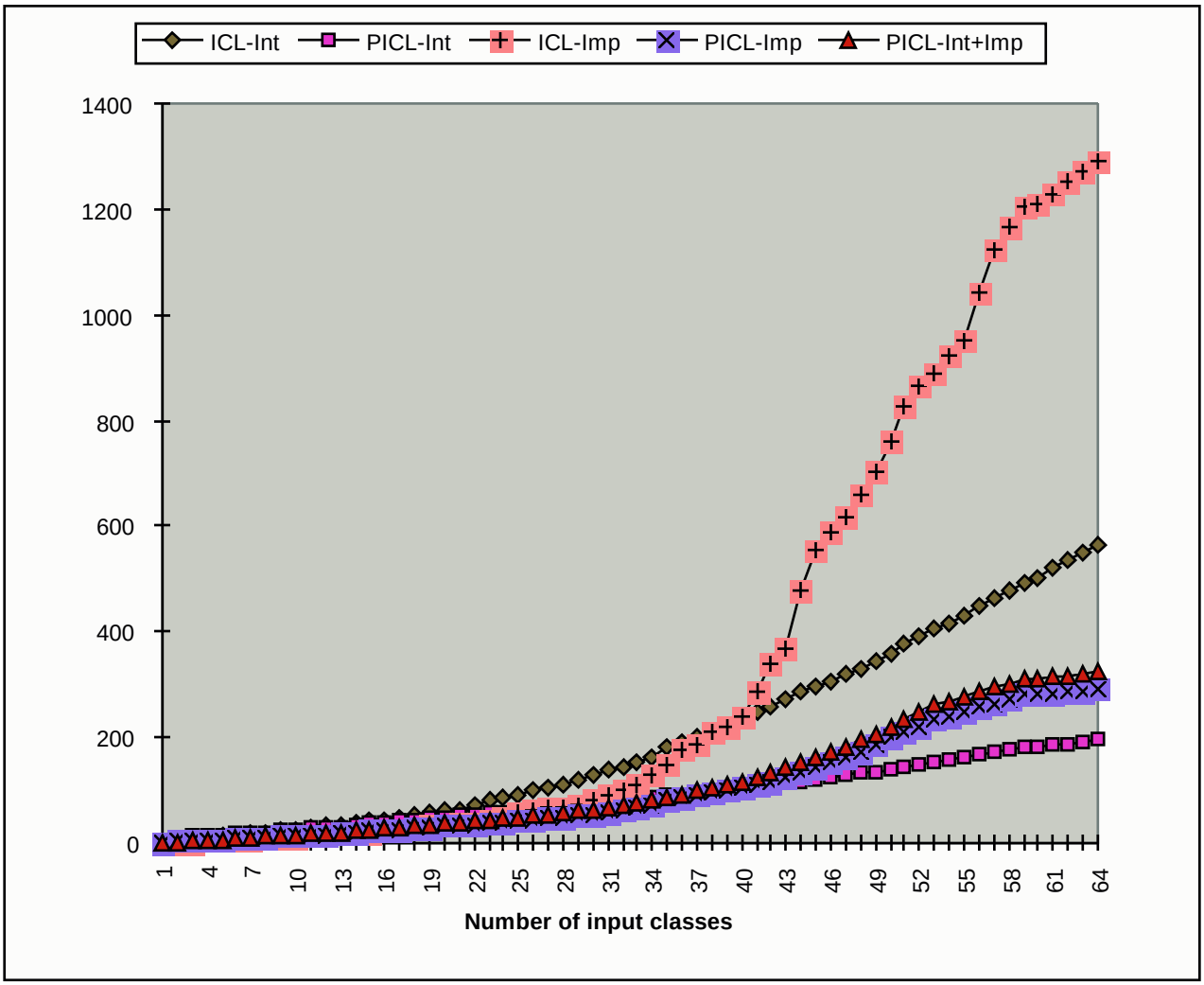

Figure 3.2. Total CPU time for incremental generation of hierarchies.

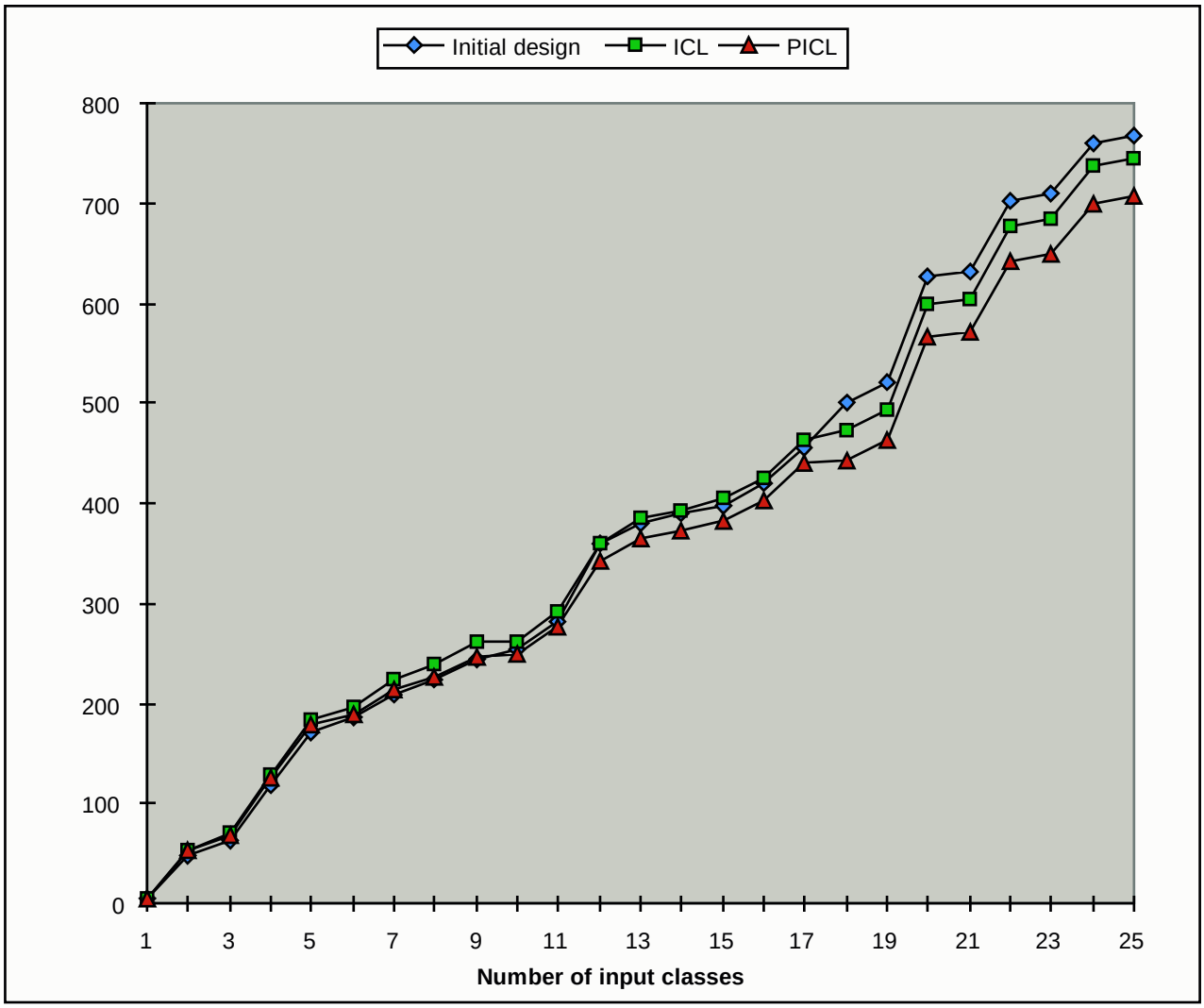

Figure 3.4. Metric *M2* for MIBs application.

4. Conclusion

Several structures useful for the analysis and design of class hierarchies were presented in the framework of the theory of concept lattices. The variants are the result of introducing inheritance, pruning of empty classes and taxonomic relations between class properties. The resulting structures guarantee that the class hierarchies are void of any redundancy and are consistent with specialization. A prototype tool for computing hierarchies based on class specifications has been developed. The tool also computes several metrics to better assess the characteristics of the hierarchies. Several experiments were performed. One experiment, described in this paper, was performed on the Collection classes of the ObjectWorks\ Smalltalk class library. Another experiment was performed on a set of class specifications from the domain of telecommunication network management application.

The experiments showed that the pruned concept (Galois) hierarchy scored best (lowest) with respect to construction time. For instance, the algorithm that we developed for computing the pruned hierarchy incrementally turned out to be much more efficient that the concept lattice incremental algorithm in the experiments. Similarly, the pruned concept hierarchy scored best (lowest) with respect the structural quality metrics. In the case of the Smalltalk class library, it appears that the reduction of complexity due to the removal of the empty classes more than offset the additional complexity corresponding to the additional multiple inheritance introduced by such removal.

These results of our experiments are encouraging, and we should continue refining our framework and experimenting with it. One interesting venue consists of trying to automate the choice of empty classes to prune from the concept lattice based on some objective criteria. At the present time they are either all kept with the concept lattice or all pruned with the pruned concept hierarchy. Between these two extremes are a large number of potential designs that can be generated by pruning only a well chosen subset of the empty concepts. At the present time, the user can interactively prune undesired classes. One avenue we are currently exploring is to try to at least partially automate the pruning process.

Another venue would try to apply the algorithms on richer class representations. Experiments on libraries from typed languages where typing information is available is one promising direction. On the other hand, the current interest in formal specifications for classes (Cheon & Leavens, 1994; Parisi-Presicce & Pierantonio, 1994) suggests that these might be used in a larger scale in the future and could be exploited by classification algorithms.

Acknowledgments

This research was partly funded by NSERC (Natural Sciences and Engineering Research Council of Canada), FCAR Funds and the Ministry of Industry, Commerce, Science and Technology of the province of Quebec, under the IGLOO project organized by the CRIM ("Centre de Recherche en Informatique de Montréal").

References

Barbut, M. & Monjardet, B. (1970). *Ordre et Classification. Algèbre et Combinatoire, Tome II*. Hachette.

Bergstein, P. & Lieberherr, K. (1991). Incremental Class Dictionary Learning and Optimization. In *Proceedings of the European Conference on Object-Oriented Programming*, Geneva, Switzerland: Springer Verlag,

Bergstein, P. L. (1991). Object-Preserving Class Transformations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications, (OOPSLA'91), Sigplan Notices*, pp. 299-313.

Booch, G. (1994). *Object-Oriented Analysis and Design* (2nd ed.). Reading, MA: Benjamin Cummings.

Carpineto, C. & Romano, G. (1996). A Lattice Conceptual Clustering System and Its Application to Browsing Retrieval. *Machine Learning*, **24**, 95-122.

Casais, E. (1991). *Managing Evolution in Object Oriented Environments: An Algorithmic Approach*. Thèse de doctorat Thesis, Geneva.

Caseau, Y. (1993). Efficient Handling of Multiple Inheritance Hierarchies. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, A. Paepcke (Ed.), Washington, DC: ACM Press, pp. 271-287.

Cheon, Y. & Leavens, G. T. (1994). The Larch/Smalltalk Interface Specification Language. *ACM Transactions on Software Engineering and Methodology*, **3**(3), 221-253.

Chidamber, S. R. & Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, **20**(6), 476-493.

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F. & Jeremaes, P. (1994). *Object-Oriented Development: the Fusion Method*. Englewood Cliffs, NJ: Prentice Hall.

Cook, W. R. (1992). Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, A. Paepcke (Ed.), Vancouver, B.C., Canada: ACM Press, pp. 1-15.

Cox, B. J. (1990). Planning the Software Revolution. *IEEE Software*, **7**(6), 25-35.

Davey, B. A. & Priestley, H. A. (1992). *Introduction to Lattices and Order*. Cambridge: Cambridge University Press.

Dicky, H., Dony, C., Huchard, M. & Libourel, T. (1994). Un algorithme d'insertion avec restructuration dans les hiérarchies de classes. In *Proceedings of the Langages et Modèles à Objets*, Grenoble:

Dicky, H., Dony, C., Huchard, M. & Libourel, T. (1996). On Automatic Class Insertion with Overloading. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, CA, USA: ACM Press, pp. 251-267.

Dvorak, J. (1994). Conceptual Entropy and Its Effect on Class Hierarchies. *IEEE Computer*, **27**(6), 59-63.

Fisher, D. H. & Pazzani, M. J. (1991). Computational Models of Concept Learning. In D. H. Fisher,M. J. Pazzani, & P. Langley (Eds.), *Concept Formation: Knowledge and Experience in Unsupervised Learning*, (pp. 3-44). San Mateo, CA: Morgan Kaufmann.

Gennari, J. H., Langley, P. & Fisher, D. (1990). Models of Incremental Concept Formation. In J. Carbonell (Eds.), *Machine Learning: Paradigms and Methods*, (pp. 11-62). Amsterdam, The Netherlands: MIT Press.

Godin, R. & Mili, H. (1993). Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, A. Paepcke (Ed.), Washington, DC: ACM Press, pp. 394-410.

Godin, R., Mineau, G., Missaoui, R., St-Germain, M. & Faraj, N. (1995a). Applying Concept Formation Methods to Software Reuse. *International Journal of Knowledge Engineering and Software Engineering*, **5**(1), 119-142.

Godin, R., Mineau, G. W. & Missaoui, R. (1995b). Incremental structuring of knowledge bases. In *Proceedings of the International Knowledge Retrieval, Use, and Storage for Efficiency Symposium (KRUSE'95)*, Santa Cruz: Springer-Verlag's Lecture Notes in Artificial Intelligence, pp. 179-198.

Godin, R., Mineau, G. W., Missaoui, R. & Mili, H. (1995c). Méthodes de Classification Conceptuelle Basées sur les Treillis de Galois et Applications. *Revue d'Intelligence Artificielle*, **9**(2), 105-137.

Godin, R. & Missaoui, R. (1994). An Incremental Concept Formation Approach for Learning from Databases. *Theoretical Computer Science, Special Issue on Formal Methods in Databases and Software Engineering*, **133**, 387-419.

Godin, R., Missaoui, R. & Alaoui, H. (1991). Learning Algorithms Using a Galois Lattice Structure. In *Proceedings of the Third International Conference on Tools for Artificial Intelligence*, S. Lee,B. Wah,N. G. Bourbakis, & W. T. Tsai (Ed.), San Jose, Calif.: IEEE Computer Society Press, pp. 22-29.

Godin, R., Missaoui, R. & Alaoui, H. (1995d). Incremental Concept Formation Algorithms Based on Galois (Concept) Lattices. *Computational Intelligence*, **11**(2), 246-267.

Godin, R., Missaoui, R. & April, A. (1993). Experimental Comparison of Navigation in a Galois Lattice with Conventional Information Retrieval Methods. *International Journal of Man-Machine Studies*, **38**, 747-767.

Godin, R., Saunders, E. & Gecsei, J. (1986). Lattice Model of Browsable Data Spaces. *Information Sciences*, **40**, 89-116.

Johnson, R. & Foote, B. (1988). Designing Reusable Classes. *Journal of Object-Oriented Programming*, **June/July**, 22-35.

Korson, T. & McGregor, J. D. (1992). Technical Criteria for the Specification and Evaluation of Object-Oriented Libraries. *Software Engineering Journal*, **March**, 85-94.

Krone, M. & Snelting, G. (1994). On The Inference of Configuration Structures from Source Code. In *Proceedings of the 16th International Conference on Software Engineering*, IEEE Computer Society Press, pp. 49-57.

Lalonde, W. R. (1989). Designing Families of Data Types Using Exemplars. *ACM Trans. on Prog. Languages and Systems*, **11**(2), 212-248.

Li, Q. & McLeod, D. (1994). Conceptual Database Evolution Through Learning in Object Databases. *IEEE Trans. on Knowledge and Data Engineering*, **6**(2), 205-224.

Lieberherr, K. J., Bergstein, P. & Silva-Lepe, I. (1991). From Objects to Classes: Algorithms for Object-Oriented Design. *Journal of Software Engineering*, **6**(4), 205-228.

Liskov, B. (1988). Data abstraction and hierarchy. *ACM SIGPLAN Notices*, **23**(5), 17-34.

Meyer, B. (1990). Tools for the New Culture: Lessons from the Design of the Eiffel Libraries. *Communications of the ACM*, **33**(9), 68-88.

Mineau, G., Gecsei, J. & Godin, R. (1990). Structuring Knowledge Bases using Automatic Learning. In *Proceedings of the IEEE Sixth Int'l Conf. on Data Engineering*, Los Angeles, CA: IEEE Computer Society Press, pp. 274-280.

Mineau, G. W. & Godin, R. (1995). Automatic Structuring of Knowledge Bases by Conceptual Clustering. *IEEE Transactions on Knowledge and Data Engineering*, **7**(5), 824-828.

Monarchi, D. E. & Puhr, G. I. (1992). A Research Typology for Object-Oriented Analysis and Design. *Communications of the ACM*, **35**(9), 35-47.

Oosthuizen, G. D., Bekker, C. & Avenant, C. (1992). Managing Classes in Very Large Class Repositories. In *Proceedings of the Tools*, Paris: pp. 625-633.

Opdyke, W. & Johnson, R. (1989). Refactoring: An Aid in Designing an Application Framework. In *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications*,

Parisi-Presicce, F. & Pierantonio, A. (1994). An Algebraic Theory of Class Specification. *ACM Transactions on Software Engineering and Methodology*, **3**(2), 166-199.

Rubin, K. S. & Goldberg, A. (1992). Object Behavior Analysis. *Communications of the ACM*, **35**(9), 48-62.

Wille, R. (1982). Restructuring Lattice Theory: an Approach Based on Hierarchies of Concepts. In I. Rival (Eds.), *Ordered Sets*, (pp. 445-470). Dordrecht-Boston: Reidel.

Wille, R. (1992). Concept Lattices and Conceptual Knowledge Systems. *Computers & Mathematics with. Applications*, **23**(6-9), 493-515.