Formal concept analysis-based class hierarchy design in object-oriented software development

Robert Godin[1], Petko Valtchev[2]

[1] Département d'informatique, UQAM, C.P. 8888, succ. "Centre Ville", Montréal (Qc), Canada, H3C 3P8

[2] DIRO, Université de Montréal, C.P. 6128, Succ. "Centre-Ville", Montréal, Québec, Canada, H3C 3J7

Abstract. The class hierarchy is an important aspect of object-oriented software development. Design and maintenance of such a hierarchy is a difficult task that is often accomplished without any clear guidance or tool support. Formal concept analysis provides a natural theoretical framework for this problem because it can guarantee maximal factorization while preserving specialization relationships. The framework can be useful for several software development scenarios within the class hierarchy life-cycle such as design from scratch using a set of class specifications, or a set of object examples, refactoring/reengineering from existing object code or from the observation of the actual use of the classes in applications and hierarchy evolution by incrementally adding new classes. The framework can take into account different levels of specification details and suggests a number of well-defined alternative designs. These alternatives can be viewed as normal forms for class hierarchies where each normal form addresses particular design goals. An overview of work in the area is presented by highlighting the formal concept analysis notions that are involved. One particularly difficult problem arises when taking associations between classes into account. Basic scaling has to be extended because the scales used for building the concept lattice are dependent on it. An approach is needed to treat this circularity in a well-defined manner. Possible solutions are discussed.

1 Introduction

An important part of object software development is the class hierarchy. The design and maintenance of such a hierarchy has been recognized as a difficult problem [1, 29]. The difficulty increases with the size of the hierarchy and the possible evolution of the software requirements that may require the incorporation of modifications in the hierarchy.

A large body of work has focused on problems related to hierarchy construction and reconstruction. Various development scenarios have been addressed (see [11]), such as:

- Building the hierarchy from scratch using:
  - objects [24],
  - class specifications [12, 8],
- Evolution of the class hierarchy to accommodate new requirements:
  - unconstrained class addition [12, 8],

    - addition constrained by backward compatibility with a previous hierarchy [28] or existing objects [16],
- Reengineering of an existing class hierarchy:
    - from the relationship between classes and their attributes/methods [2, 4],
    - using code analysis tools [7, 15],
    - by applying refactorings [27, 9],
    - from UML models including associations [19],
    - from access patterns in applications [32],
    - prompted by detecting defects using software metrics [30],
- Reengineering procedural code into an object environment [31, 34],
- Merging existing hierarchies [33].

In many cases, the proposed approaches rely on algorithms that are not grounded on well-established theoretical results. Thus, the corresponding methods may yield unpredictable results. In some cases, the exact form of hierarchies depend on adjustable parameters of the procedures. In contrast, Formal Concept Analysis (FCA) provides a natural theoretical framework for class hierarchy design and maintenance and several researchers have adopted this framework ([36, 32, 7, 27, 17, 12]). Hierarchies produced within this framework have a well-defined semantics that remains independent from the concrete algorithms used. In addition, the produced hierarchies tend to conform to general quality criteria such as *simplicity*, *comprehensibility*, *reusability*, *extensibility* and *maintainability*.

These high-level criteria represent desirable features of the final result that very much depend on its usage during further stages of the software process. However, these high-level criteria are knowingly favored by two more concrete quality criteria that may be measured directly on the target software artifacts:

1. *Minimizing redundancy*. Having each artefact defined in one single place in the code/specifications is a well-known software design principle that a class hierarchy should promote [20, 21, 9]. In contrast, keeping several definitions of the same artefact at possibly different locations may lead to inconsistencies between copies. Moreover, redundancy increases the complexity of the resulting software and, more dramatically, speaks about possible flaws in the design since repeating code/specification chunks is a hint that these have not given rise to the appropriate abstractions that help embed them into a single software unit. Besides, lessons from building large class libraries [26] 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 hierarchies are sometimes created for code reuse purposes, especially those in code libraries. Thus, the inheritance between classes in the hierarchy my not correspond to any particular reality in the corresponding domain but rather help optimize code sharing in the hierarchy. However, as observed by [5], in the long run such a designing free of semantic concerns may produce libraries that are difficult to understand and hence to reuse. Therefore, many authors have advocated the enforcement of consistency with specialization in inheritance hierarchies ([20, 25, 22, 2, 4, 3]) in particular, in order to achieve better comprehensibility and reusability.

Hierarchies produced by methods based on FCA are guaranteed to meet these criteria. Depending on the design goals and available specifications, several alternative hierarchy types may be considered within this framework. These hierarchies can be viewed as ideal structures similar to relational database normal forms, with each normal form addressing a particular design goal.

In the following, a set of normal forms for class hierarchy design is described, all of them based on the FCA framework. These normal forms synthesize the previous propositions in a unified framework. Section 2 introduces the basic idea by defining the attribute factored lattice form and relating it to a concept lattice. Section 3 introduces the more compact attribute factored subhierarchy form which is based on the set of object and attribute concepts of the concept lattice. Section 4 proposes normal forms for factoring methods taking into account the distinction between signature and body and the possibility of method redefinitions. Section 5 discusses the factoring of associations and the complications introduced by circular dependencies. Available software tools are listed in Section 6 whereas Section 7 provides an overview of some on-going industrial projects involving FCA and lattices.

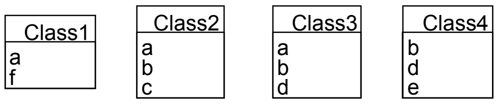
2 Attribute factored lattice form

Fundamental constructs of object software are the notions of *object* and *class*. A class is an abstraction for a set of objects that share the same characteristics. In programming languages, these characteristics, also called *members* of the class, are *attributes* and *methods*. In modelling languages, *associations* are also used to relate classes. An *attribute*, also called instance variable or data member, contains data used to model the state of an object. This section is concerned with the attributes of the classes. Methods and associations will be examined in the following sections.

When using formal concept analysis for class hierarchy design, the set of formal objects G is a set of software artefacts, i.e., classes, objects or program variables, which are used as a starting point in the search for a suitable class hierarchy. The set of formal attributes M corresponds to properties of the classes or objects. Relevant properties include attributes (instance variables), methods (body and/or signature of the method) or associations (in the case of classes). Further information may be available such as values of the variables in objects or links to specific objects for associations. In this paper, we only consider the case where the starting point is a set of class specifications, i.e., G is a set of classes. Nevertheless, the principles are directly transposable to the case of example objects or program variables.

First, we consider the case of factoring out attributes of the classes. Let's take a simple example to illustrate the basic idea. Suppose that we have a specification of the attributes for a set of four concrete classes as illustrated in Figure 1. The specification could be interpreted as the exact set of concrete classes that the hierarchy must contain, i.e., these classes will be the only ones to "produce" objects in an application. Other classes of the hierarchy can be used to factor common specifications.

This input specification may be produced in several ways depending on the development scenario. For example, with a forward engineering process, classes and their attributes are first specified in the analysis phase of the process. Thus, they are produced by the analyst and typically expressed by means of a modeling language such as UML (as in Figure 1 on the left). Within a re-engineering process, the classes are already organized in a possibly larger hierarchy, with their respective specifications spread over the entire set of classes in the hierarchy. In this case, the attribute set of each concrete class is compiled from all its super-classes in the hierarchy. The goal of the corresponding reengineering scenario is now to refactor an existing hierarchy, i.e., to suggest a different organization of specifications within a new set of classes while preserving the semantics of the initial hierarchy. The semantics here is limited to the behavior of objects from all concrete classes, an approach that allows the modifications in the already existing source code that uses services from the initial hierarchy to be kept to a minimum. The incidence relation I of the formal context $\mathbb{K}$ representing the



	a	b	c	d	e	f
1	x					x
2	x	x	x			
3	x	x		x		
4		x		x	x	

Fig. 1. Left: Example specification; **Right**: Corresponding context.

set of four classes and their instance variables is shown in Figure 1 on the right. The context is drawn as a cross table with classes identified by integers and the variables by letters.

As the problem is to organize these classes in a hierarchy, a concept lattice is used as a guideline for the design of such a hierarchy (in some sense, it provides an ideal design). To that end, each formal concept is interpreted as a class of the hierarchy. Moreover, the sub-concept relation links are seen as specializations between classes. Figure 1 (on the left) shows the line diagram of the concept lattice with a *reduced labeling* of concepts. The labels assigned to the concepts indicate where, i.e., in which class, a particular attribute should be declared. For example, the attributes a and b will have to be placed at two general classes that are located immediately below the root class of the hierarchy. It is noteworthy that for class hierarchies, the bottom concept is dropped since it is of no use.

Figure 2 shows the *attribute factored lattice form* hierarchy that corresponds to this interpretation of the concept lattice. The four initial classes remain in the hierarchy but there are fewer declared attributes in these classes because of the factoring produced by the concept lattice. New classes (classes 5 through 9) are added that factor out common attributes. These are abstract classes because instances are created only for the four initial classes. The nature of the reduced labeling of the concept lattice guarantees that each attribute appears exactly once in the hierarchy. Object attributes in the initial concrete classes remain unchanged. However, part of them are now inherited from some new classes. Globally, all subclasses are specializations since they inherit the attributes of parent classes with no exception. There are no cancellations.

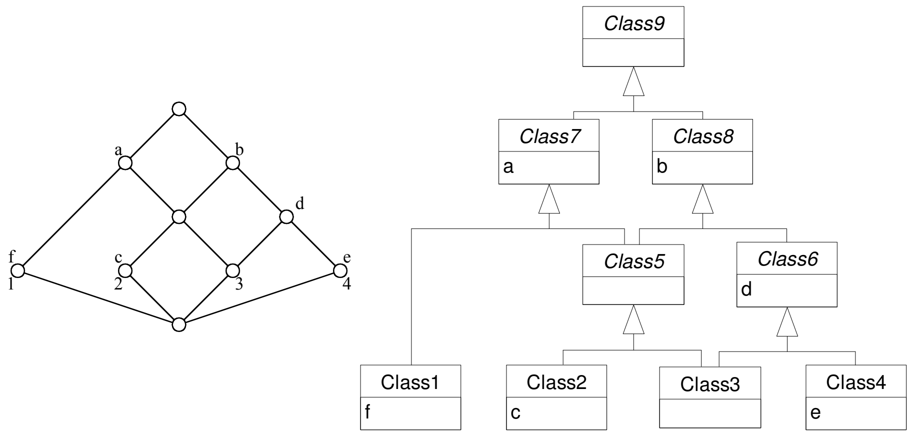


Fig. 2. Attribute factored lattice form for the input specification of Figure 1.

From a client point of view, using this hierarchy will produce the same effect as using the initial four classes. Therefore the generated hierarchy can be interpreted as a refactoring of the initial four class specifications.

There is a large number of possible designs that can minimize redundancy. The concept lattice attains this goal minimizing the number of classes and the amount of multiple inheritance, which is often considered as undesirable since more complex. This is achieved by grouping attributes in classes whenever possible, as illustrated by the following example. Figure 3 shows two input classes. The attribute factored lattice form that appears in Figure 4 on the left factors out the common attributes a and b in the new `Class3`. The design presented in Figure 4 on the right also factors out the common attributes but is unnecessarily complex since it contains two classes, one for each attribute, thus capturing classes 1 and 2 in a multiple inheritance pattern. In contrast, the design in Figure 4 on the left is simpler while still providing the same quality criteria of redundancy avoidance and conformance to specialization.

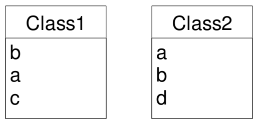


Fig. 3. Input specification.

An important hypothesis underlying the approach is that identical attribute names identify properties that can be matched from a semantic point of view. If the matching is based on attribute names, care should be taken to identify semantic commonalities and rename attributes as necessary.

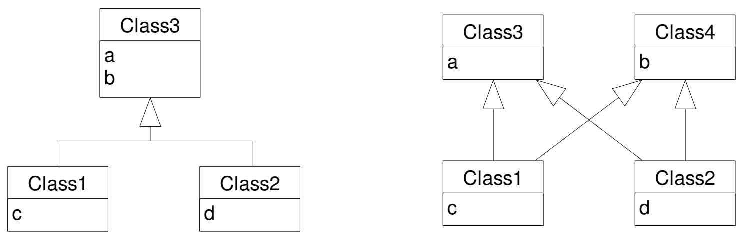


Fig. 4. Attribute factored lattice form for the input specification of Figure 3 and alternate factoring.

3 Attribute factored subhierarchy form

The concept lattice is an exhaustive representation of commonalities among a set of concrete classes. As its size could grow rapidly, one may think of skipping some of its nodes to keep the whole structure manageable. Thus, a first idea could be to remove abstract classes that declare no properties. These classes, often called *empty classes*, can be removed without violating the formal quality criteria, i.e., no redundancy and specialization[3]. In the example in Figure 2 on the right, the empty classes, `Class5` and `Class9`, could be omitted (see Figure 2). Even though `Class3` declares no attributes, it has to be kept because it is not abstract.

The structure that occurs after the removal of all empty classes, called Galois subhierarchy in [7], corresponds to the set of all *attribute* and *object concepts* of the concept lattice. We recall that given an attribute, its *attribute concept* is the maximal concept in whose intent the attribute appears. Intuitively, the attribute concept of an attribute a is labeled by "a" in the diagram with reduced labeling. The notion of *object concept* is dual, i.e., object concepts have at least one object label. When re-engineering a class hierarchy within the FCA framework, the set of attribute and object concepts constitutes the minimal part of the concept lattice that should be preserved in order to satisfy both concrete formal quality criteria while respecting the initial class specification. In fact, object concepts have to be kept because they correspond to the concrete classes that are used by client code, in particular because they are the classes that can be instantiated[4]. Attribute concepts are in turn necessary because they correspond to classes that declare attributes which are further inherited by their subclasses.

The class hierarchy produced from the Galois subhierarchy constitutes what we call the *attribute factored subhierarchy form*. The class hierarchy for our example is shown in Figure 5 on the right.

Compared to the lattice, the resulting structure has fewer classes while still preserving the quality criteria. Between the lattice form and subhierarchy form, many alternative designs can be produced by selectively keeping subsets of the empty classes.

[3] Here "formal" is used in the sense of measurable, as opposed to "informal" quality criteria, e.g., comprehensibility, which are hard to measure.
[4] For environments that support multi-instanciation, the concrete classes could also be omitted.

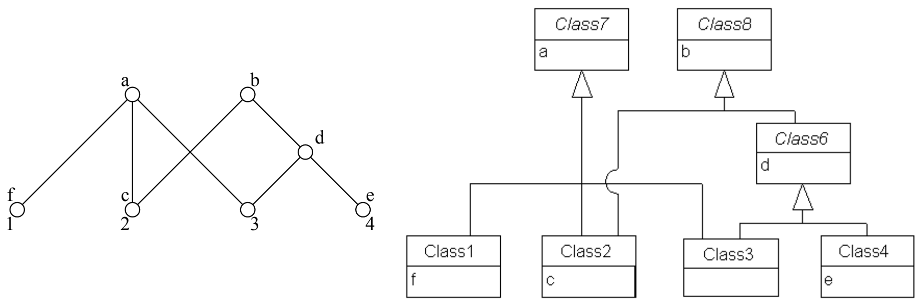


Fig. 5. Galois subhierarchy for the context in Figure 1 and corresponding attribute factored subhierarchy form.

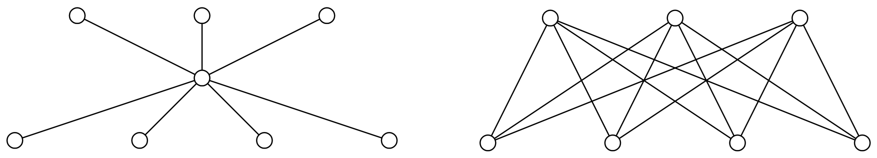


Fig. 6. Lattice and subhierarchy forms.

These intermediate designs also preserve the quality criteria. There are no clear cut obvious formal criteria for assessing the usefulness of the empty classes. However, there are some cases where they clearly have value. For example, when a large set of classes inherit from another set as in the subhierarchy form in Figure 6 on the right, the intermediary empty class of the lattice form simplifies the design in the sense that it reduces the number of multiple inheritance situations. Indeed, multiple inheritance is knowingly more complex and is not necessarily allowed by all object programming languages. Therefore, it may sometimes be undesirable in the class hierarchy design. To return to our example in Figure 6, in the worst case, the removal of an empty class results in all possible direct links between parent and child classes. Therefore, $n + m$ specialization relationships in the lattice, with n and m being the number of parent/child classes, respectively, might be replaced by nm such relationships in the subhierarchy.

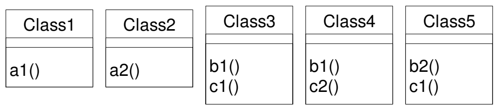
The above situation is an extreme case and need not to occur each time. When n and m are small, the value of the empty class is less obvious. Moreover, the value nm is an upper bound for the effective number of links that need to be created. In many cases a pair of parent and child classes will not create a new link since both classes are already linked through an alternative path of links in the hierarchy. For example, the removal of the empty `Class5` in the lattice form of Figure 2 does not require a new link between its parent `Class3` and its child `Class8` in the subhierarchy form of Figure 2 because attribute b of `Class3` is also inherited from the path going through `Class6`. Some work has been done on guiding the choice of empty classes using class hierarchy metrics [14].

An important feature of the above hierarchical normal forms is that usually they produce multiple inheritance. However, this does not automatically mean that the approach is useless for single inheritance environments. In fact, the normal forms represent an ideal structure that can be used as a starting point from which a good single inheritance hierarchy can be extracted. For example, to reduce multiple inheritance to a single one, a common practice is to choose a single parent class to keep in the hierarchy while replacing the remaining links by delegation references. A less elegant, but sometimes unavoidable, strategy for eliminating multiple parents is to duplicate locally the information that is inherited from the disconnected parents.

4 Method factored forms

Another important part of class hierarchies in OO development are the behavioral specifications incorporated in the class descriptions in the form of methods. Method specifications may be divided into two parts. Method signatures specify the way a method is invoked (name, parameters, return type), while the actual processing carried out by a method is specified by its body. The entire set of method signatures for a given class, also called its *interface*, is usually considered as its contract, i.e., the set of services the class must offer. If we want simply to factor out the method bodies, we can use the same approach as for attributes.

Figure 7 shows on the left an example of input specifications for five classes and the methods they support. Objects of `Class3` need to ”respond” to calls invoking methods `b1()` and `c1()`. Here, methods whose names begin with the same letter, i.e., a, b, or c, share the same signature while implementing it in different bodies (but we shall ignore this for now and come back to it later in the text). Figure 7 shows the corresponding context on the right.



	$a1$	$a2$	$b1$	$b2$	$c1$	$c2$
1	x					
2		x				
3			x		x	
4			x			x
5				x	x	

Fig. 7. Example specification and corresponding context.

As for the attributes case, the concept lattice of this context reveals an organization of the class hierarchy that guarantees no redundancy and conformance to specialization. The reduced labeling in Figure 8 on the left indicates the location in the hierarchy where each method body should be specified. Shown in Figure 8 on the right is the corresponding class hierarchy called *method body factored lattice* form where each concept is interpreted as a class. Here again, when omitting the empty classes, we obtain the *method body factored subhierarchy* form. In our example, the difference with the lattice form is made up of the abstract `Class8` which is skipped in the subhierarchy.

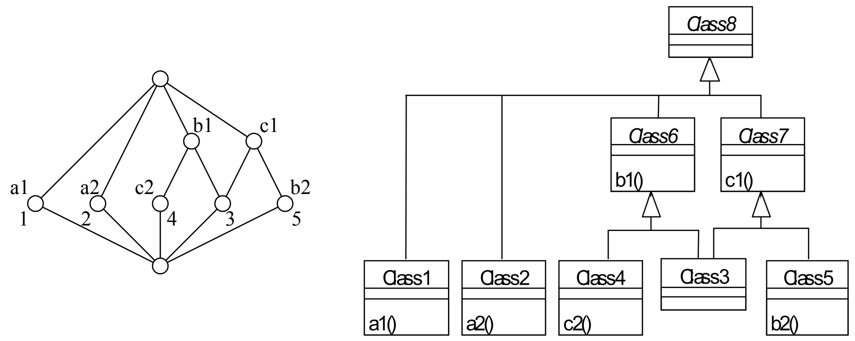


Fig. 8. Left: Reduced labeling of the concept lattice for the context in Figure 7; **Right:** Method body factored lattice form of the hierarchy.

With method factoring that distinguishes method signatures from their bodies, the computation gets more complex. In our example, methods sharing the same letter in their names represent different method bodies, i.e., implementations, of the same method signature. For instance, methods `a1()` and `a2()` represent two different implementations of the `a()` signature. Such a distinction is sensible here since in many object-oriented development environments and languages, the signature and body of a method can be declared separately and there may be more than one body for the same signature. For example, in Java, a method can be declared as a mere signature (an *abstract* method) while leaving one or more implementations as a responsibility for the subclasses that not only inherit the signature but also need to effectively carry-out the specified work. Under this circumstance, it becomes necessary to determine the class where each aspect (signature and body) will be declared in the hierarchy.

The appropriate FCA constructs that help formalize the factoring of methods including their signatures are many-valued contexts and conceptual scaling. Thus, for each method signature `m`, we define a many-valued attribute `m`. In our example, there are three many-valued attributes for the three signatures `a`, `b` and `c` (see Figure 9 on the left). The values of a many-valued attribute are the method body names. The values `a1` and `a2` represent method bodies for `a`.

Figure 9 on the right, shows the scale $\mathbb{S}_a$ for the many-valued attribute a. The corresponding concept lattices for each scale are illustrated in Figure 10. The special value a in the scale for the multi-valued attribute of the same name represents the declaration of the method signature. The values for the different bodies are children nodes of this special value in the scale.

The general case of method `m()` with bodies `m1()`, `m2()`, `...`, `mn()` is illustrated in Figure 11. The scale $\mathbb{S}_m$ is built by adding attribute m to a nominal scale for scale objects `m1`, `m2`, `...`, `mn`. Attribute `m` is assigned to all scale objects in order to represent

	a	b	c
1	$a1$		
2	$a2$		
3		$b1$	$c1$
4		$b1$	$c2$
5		$b2$	$c1$

$\mathbb{S}_a$	a	$a1$	$a2$
a	x		
$a1$	x	x	
$a2$	x		x

Fig. 9. Left: Many-valued context representing the shared method signatures; **Right**: Scale $\mathbb{S}_a$ for the `a` method signature.

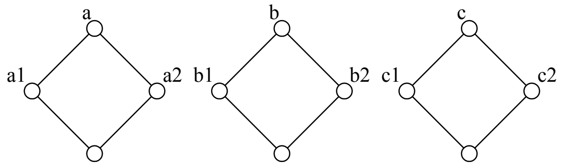


Fig. 10. Concept lattices of the scales for the three method signatures, `a`, `b` and `c`.

the fact that every method body `mi()` implements signature `m()`. Later, we will show how more general scales are used when taking method redefinitions into account.

$\mathbb{S}_m$	m	$m1$	$m2$	$\ldots$	mn
m	x			$\ldots$	
$m1$	x	x		$\ldots$	
$m2$	x		x	$\ldots$	
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
mn	x			$\ldots$	x

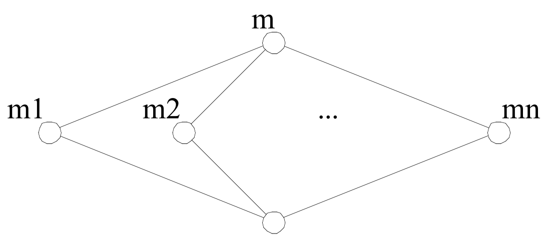


Fig. 11. Left: Scale $\mathbb{S}_m$ for the general case; **Right:** Concept lattice for the scale $\mathbb{S}_m$.

The one-valued context derived from the scaling for method signatures with our example appears in Figure 12.

The reduced labeling of the concept lattice (see Figure 9 on the left) produced from the derived one-valued context shows where each method signature and body parts should be declared. The corresponding class hierarchy called *method signature/body factored lattice* form is illustrated in Figure 14 on the right. The notation `m()` in the UML diagram represents the declaration of the signature while `mn()` represents the declaration of a method body corresponding to the `m()` signature. Here again, the class hierarchy is guaranteed to have no redundancy because each method signature and body is declared exactly once. Once again, the resulting class hierarchy conforms to special-

	$\mathbb{S}_a$			$\mathbb{S}_b$			$\mathbb{S}_c$		
	a	$a1$	$a2$	b	$b1$	$b2$	c	$c1$	$c2$
1	x	x							
2	x		x						
3				x	x		x	x	
4				x	x		x		x
5				x		x	x	x	

Fig. 12. One-valued context derived from the many-valued context in Figure 9 after scaling.

ization. As previously, in the method signature/body factored subhierarchy form, all empty classes are dropped out.

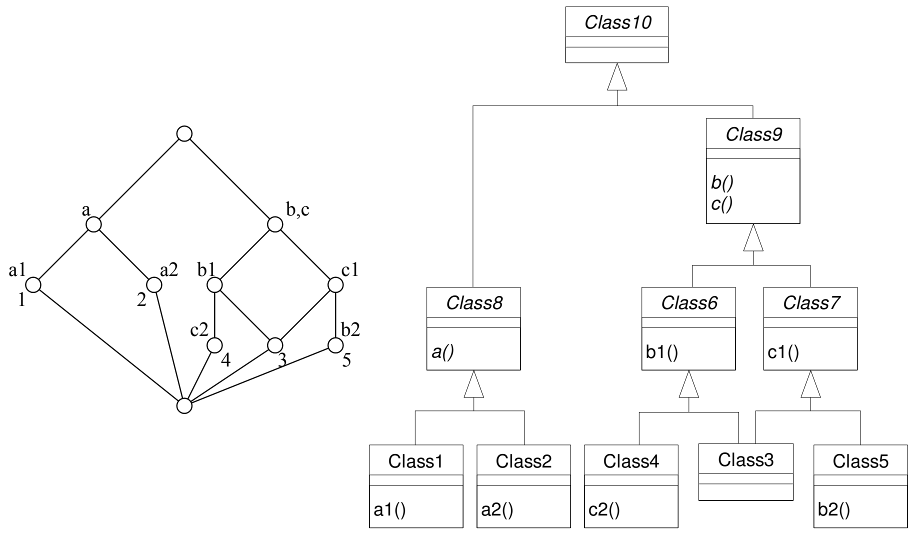


Fig. 13. Concept lattice of the derived one-valued context of Figure 12 and its method signature/body factored lattice form.

Finally, we consider the method factoring problem in its most general settings, i.e., when overriding, or redefinition of inherited methods, is allowed. For example, in most object languages, a class `Class1` which inherits a method `a2()` with a signature `a()` from a class `Class2` could still declare its own method `a1()` of the same signature `a()`. In this case, `a1()` overrides `a2()` in the sense that only `a1()` is directly available for objects from `Class1`. More precisely, the invocation of the signature `a()` on an object of `Class1` will in fact call method body `a1()` while `a2()` remains hidden for objects of `Class1`. Such redefinitions should conform to specialization. More sophisticated redefinitions can also be supported where the signatures are not identical.

Notice that our framework is orthogonal to the type of redefinitions that are supported (e.g. *covariant* or *contravariant*).

As an illustration, suppose that in the previous example, signature `b` is a specialization of signature `a` and method body `b2` is a specialization of `b1`. If the development environment supports redefinitions, this knowledge can be used to produce a hierarchy with finer factoring. For this purpose, we simply use an enhanced scale taking into account the relationships induced by the specialization order among method signatures and bodies. For our example, the relationships between the methods for the `a` and `b` signatures are represented in the scale of Figure 14.

$\mathbb{S}_{ab}$	a	$a1$	$a2$	b	$b1$	$b2$
a	x					
$a1$	x	x				
$a2$	x		x			
b	x			x		
$b1$	x			x	x	
$b2$	x			x	x	x

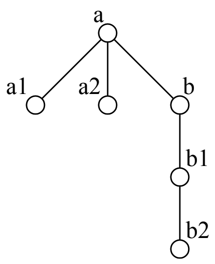


Fig. 14. Scaling and graph, both representing the specialization relationships for `a` and `b`.

In addition, the scale of Figure 15 reflects the fact that method `c2()` is a specialization of method `c1()`. The graphical representation of the concept lattices of these scales is given on the right part of both Figures 14 and 15.

$\mathbb{S}_c$	c	$c1$	$c2$
c	x		
$c1$	x	x	
$c2$	x	x	x

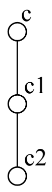
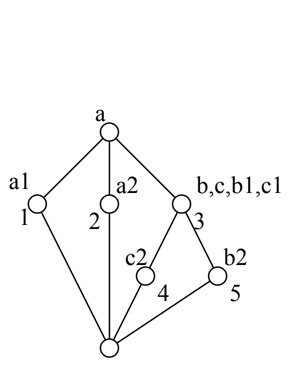


Fig. 15. Scaling and graph, both representing the specialization relationships for `c`.

The concept lattice produced by applying these scales and the corresponding method redefinition lattice form class hierarchy are shown in Figure 16. The resulting hierarchy reveals where each method signature and body should be declared without redundancy and in conformance with specialization. Again, one could consider omitting empty classes.

When taking specialization relationships between redefined properties into account, absence of redundancy is formalized by the notion of maximal factorization [7].



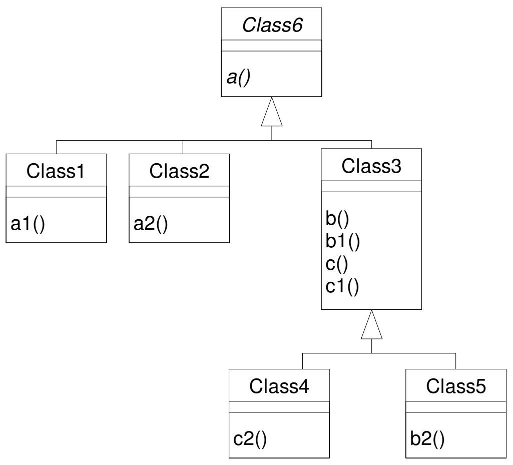


Fig. 16. Concept lattice after applying the scaling of Figures 14 and 15, and the corresponding method redefinition factored lattice form.

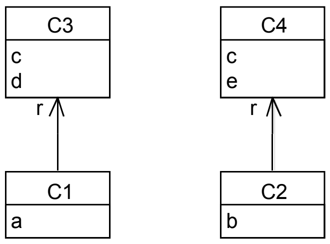
A class hierarchy is *maximally factorized* iff whenever two properties p_1 and p_2, which are declared by the classes C_1 and C_2, respectively, have a least upper bound in the specialization hierarchy of the properties, say p_3, then there is a common superclass C_3 of C_1 and C_2 declaring p_3. The special case of $p_1 = p_2 = p$, implies that p is declared in C_3.

A hierarchy produced from scaling based on the order relationships between properties is guaranteed to be maximally factorized [7]. For example, observe that `a1()` and `b1()` have `a()` as an upper bound in the scale. In the class hierarchy, the signature `a()` is declared in `Class6` which is a superclass of the classes where `a1()` and `b1()` are defined, i.e., `Class1` and `Class3`.

5 Factoring associations

Many environments for object-oriented analysis and design admit explicit representations of inter-class *associations* and these are an important part of the UML description arsenal. Associations are to be seen as a generic expression of the links that connect individual objects, e.g., kinships, spatial and time relations, part-of relations, etc. Most of the time, they correspond to a classical binary relation between the objects in the extensions of the related classes. For example, the UML model of Figure 17 shows an association between class `C1` and class `C3` and another one between class `C2` and `C4`. For the purpose of illustration, the classes also have some attributes.

Factoring associations can also be considered in the design process as inheritance spreads over associations too [19]. In fact, the specialization among associations appears naturally as most of the time associations models admit specialization links among associations. Moreover, in UML, associations can have their own properties. However, for the present discussion, we take a simplified model of an association: associations



	a	b	c	d	e	r
1	x					3
2		x				4
3			x	x		
4			x		x	

Fig. 17. Example UML model with associations and corresponding context with a formal attribute for the UML association role.

are directed and their only descriptor is a name. Thus, our model corresponds to what is called an *association role* in UML, which allows them to be further assimilated to object attributes whose values are other objects. This is similar in spirit to representing associations by object-valued attributes, or references, in object languages. Thus, a UML association role may be represented in a context by introducing a many-valued attribute named by the corresponding role. Such a transformation yields the context of Figure 17 with the many-valued attribute `r`. It is noteworthy that the classical object attributes are represented as formal attributes of the context, as before. In contrast, the values of the many-valued attribute `r` are actually the identifiers of the classes that are pointed to by the association. Thus, the numbers 3 and 4 stand for the classes `C3` and `C4`, respectively. In terms of FCA, this means that the values of the many-valued attribute correspond to formal objects.

This introduces a circular dependency of the context on itself since the only way of constructing a lattice out of it is to scale the attribute `r` whose domain is (a part of) the context. Logically, such a scale would require a conceptual structure to be built on top of the context in order for meaningful abstractions to be made available as scaling targets. However, the construction of a meaningful structure is exactly what is the global analysis process is aimed at. In summary, to construct the conceptual hierarchy of the context, there must be another hierarchy on the same context to play the role of a scale and any consistent processing would reasonably require both hierarchies to be identical. As indicated in [35], the resulting apparent deadlock could be successfully resolved by a simple bootstrapping strategy. More precisely speaking, the proposed approach applies an iterative procedure that alternates lattice constructions and scaling. At each iteration, the lattice resulting from the previous iteration is used as a scale that helps enrich the current context and therefore leads to a more precise lattice at the next step. The process halts in a finite number of steps with a lattice which remains stable along two consecutive steps. The iterative approach is illustrated in the following.

In a first iteration, we consider a nominal scale $\mathfrak{J}$ for the `r` association role. The resulting one-valued context and the concept lattice noted $\mathbb{B}^0$ appear in Figure 18.

Interpreting the concept lattice as a class hierarchy produces the design in Figure 19. The first iteration generates a new superclass `C5` for `C3` and `C4` based on the recognition of their common attribute `c`. This new class would have been produced by factoring attributes alone and therefore does not bring value to the classical FCA-based factor-

						$\mathbb{S}_r = \mathfrak{J}$	
	a	b	c	d	e	3	4
1	x					x	
2		x					x
3			x	x			
4			x		x		

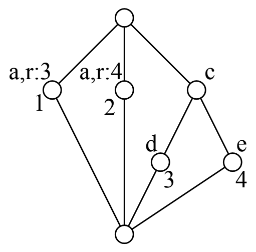


Fig. 18. One-valued context derived by a nominal scale for the association role `r` and its concept lattice $\mathfrak{B}^0$.

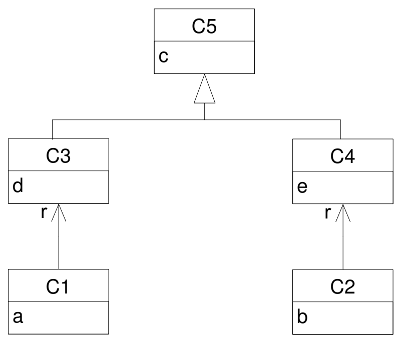


Fig. 19. Class hierarchy from $\mathfrak{B}^0$ after first iteration.

ing. However, by taking associations into account, we can go further by factoring the association role. Thus, given the `r` role from `C1` to `C3`, we can infer that there is also an association role `r` from `C1` to `C5` because `C3` is now a subclass of `C5`. The same is true for the `r` role from `C2` to `C4`. This can be taken into account by enriching the first context with a scale that incorporates the superclass relationships discovered in the concept lattice $\mathbb{B}^0$ of the first iteration context. The result is the second context which yields the concept lattice $\mathbb{B}^1$, both shown in Figure 20.

This leads to the discovery of a common association role abstracted in a new concept labeled `r:5`. This new concept produces the new class `C6` in the new hierarchy of Figure 21, which factors the common association role referencing `C5`. Again, the newly discovered class is used to enrich the second iteration context by incorporating it in the scale. In this way, the third iteration context arises. In our example, the resulting hierarchy is isomorphic to the previous one, thus yielding a fixed point of our iterative process. This constitutes the final design whereby the resulting fixed point is called the *the association factored lattice form*.

The above simplified procedure has been applied to the re-engineering of UML analysis models, i.e., UML class diagrams with a rich set of descriptors. Taking into account all those descriptors that translate relevant aspects of both classes and associations in a UML class diagram, e.g., association multiplicity factors, property visibil-

						$\mathbb{S}_r$		
	a	b	c	d	e	3	4	5
1	x					x		x
2		x					x	x
3			x	x				
4			x		x			

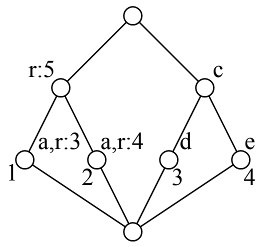


Fig. 20. Second iteration context enriched with the scale derived from the first iteration concept lattice $\mathbb{B}^0$ and its concept lattice $\mathbb{B}^1$.

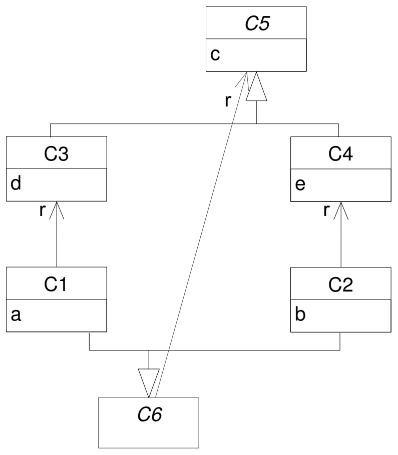


Fig. 21. Second iteration hierarchy.

ity, etc., requires a full-scale translation of the software object landscape elements into FCA. As a suitable representation, the notion of *Relational Context Family* has been proposed which may be thought of as a cluster of formal contexts whose formal objects are linked by a set of binary relations. A detailed presentation of the relational framework in FCA may be found in [18]. The application of the framework to the analysis of UML class diagrams and the underlying Iterative Cross-Generalization (ICG) method is described in [6].

6 Tools

The main algorithmic challenge of the FCA-based hierarchy design is the construction of the Galois subhierarchy. A number of methods for this task have been designed, starting with the work of Godin and Mili [13], followed by the publication of the algorithms ARES [7], AISGOOD [10], and *Ceres* [23]. It is noteworthy that most of the published methods are incremental procedures, i.e., they construct the hierarchy by acquiring the input data, e.g., the classes, one-by-one and integrating each newly inserted class into the current structure. A summary of the methods for class restructuring that do not rely on a FCA results may be found in [17].

Most of the methods for Galois subhierarchy manipulation have been designed to work on software-related datasets. The authors have provided generic implementations, however there is no code repository of all the original implementations. Instead, recently, the implementation of a generic platform for FCA and further lattice manipulations, called GALICIA[5], has been launched.

GALICIA is intended as an integrated software platform including components for the key operations on lattices and related partially ordered structures such as the Galois subhierarchy that might be required in practical applications or in more theoretically-oriented studies. It was designed to cover the whole range of basic tasks that make up the complete life-cycle of a lattice/subhierarchy: data input, construction and visualization. The platform is implemented in Java. On the algorithmic side, GALICIA includes conform implementations of the major Galois subhierarchy methods that are often accompanied by a set of experimental versions. Moreover, an entire component of the platform is dedicated to the ICG framework that produces several subhierarchies on a set of mutually related formal concepts representing a UML class diagram.

7 Recent applications

One of the recent and promising application of the FCA-based methods for class design has been carried out within the MACAO[6] project. MACAO is a joint project of France Télécom, LIRMM, and SOFTEAM[7], a French software company specialized in CASE tool development. It is aimed at enhancing the Objecteering[8] CASE tool, an ”all-in-one” environment that combines the Eclipse[9] development environment with model support (via full UML compliance) and code generation.

As part of the project goals, the ICG procedure within GALICIA has been connected to Objecteering. ICG thus provides to Objecteering users, i.e., software developers, the possibility to analyze the UML class models they have created within the CASE tool and to receive valuable suggestions as to possible improvements in these models. Operationally speaking, the UML model from the main tool is exported as a RCF and loaded into GALICIA. The result of ICG running on the RCF, once translated back into UML is fed into Objecteering. The initial and the re-engineered diagram can then be compared and the differences are evaluated.

The Objecteering - ICG tandem has been experimentally applied to a set of existing models of France Télécom, including medium-sized (e.g., a common user data model for several telecom services) and large-sized ones (e.g., a design model of an information system). The user feed-back about the relevance of the suggested new classes and

[5] See the website at: `http://www.iro.umontreal.ca/∼galicia`.
[6] http://www.lirmm.fr/∼macao.
[7] http://www.softeam.fr/
[8] http://www.objecteering.com/
[9] http://www.eclipse.org/

associations was positive, as the ICG tool has discovered many abstractions that would be difficult to extract manually.

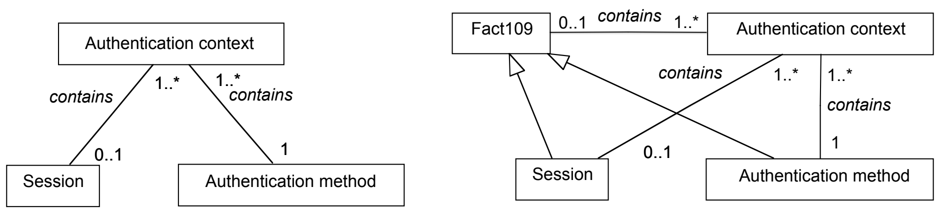


Fig. 22. Example of the creation of a new association (adapted from [6]).

Figure 22 depicts a part of the class diagram in one of the projects that was included in the study. The left hand side shows the initial diagram, whereas the right hand side represents the ICG suggestion. The remarkable element is the abstraction of a new association out of the initial associations named `contains`. The creation of the new association has further led to the discovery of a new class, `Fact109`. For a more detailed description of the experimental results for ICG, the reader is referred to [6].

8 Conclusion

We have presented a set of normal forms for the class hierarchy design problem in object oriented development. Each normal form addresses the factoring of different aspects of class properties based on the FCA framework. Although the factoring of attributes, methods and associations was presented separately, they could evidently be combined. The ultimate normal form, called fully factored lattice/subhierarchy form, consists in factoring out every aspect of the class specifications: attributes, method signatures/bodies with redefinition and associations.

These normal forms can be used as guides for the design of class hierarchies within several development scenarios. They could be incorporated in integrated development environment tools by automating the generation of the normal forms. A large body of algorithmic procedures are available to produce the underlying concept lattices efficiently. In practice, as is the case of normal forms for database design, the class hierarchy normal forms should be considered as ideal structures from which some deviations might be considered based on considerations that are not taken into account in the normalization process. Within tool support, we consider that the design process should not be seen as completely automated and some form of user interaction should be provided to produce the final hierarchy possibly by taking a normal form as a starting point or by contrasting some existing design with a normal form, thus revealing potential design anomalies.

Acknowledgment

The most recent research described in the paper was made possible by the respective Canadian NSERC (National Science and Engineering Research Council) grants hold by the authors as well as the team grant from FQRNT (Fonds de Recherche sur la Nature et les Technologies) of Quebec. The authors would like to thank as well their colleagues Rokia Missaoui and Marianne Huchard for the fruitful discussions and critical remarks. Thanks go to all the members of the software engineering team behind the Galicia platform.

References

1. G. Booch. *Object Oriented Analysis and Design with Applications, Second Edition*. Addison-Wesley, 1994.
2. E. Casais. *Managing Evolution in Object Oriented Environments : An Algorithmic Approach*. PhD thesis, Université de Genève, 1991.
3. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, and P. Jeremaes. *O-O Development — The FUSION Method*. Prentice Hall, 1993 1993.
4. W.R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In A. Paepcke, editor, *Proceedings of the Xth OOPSLA*, pages 1–15. ACM Press, 1992.
5. B.J. Cox. Planning the Software Revolution. *IEEE Software*, 7(6):25–33, November 1990.
6. M. Dao, M. Huchard, M. Rouane Hacene, C. Roume, and P. Valtchev. Improving Generalization Level in UML Models: Iterative Cross Generalization in Practice. In H. Delugach K. E. Wolff, H. Pfeiffer, editor, *Proceedings of the 12th Intl. Conference on Conceptual Structures (ICCS'04)*, volume 3127 of *Lecture Notes in Computer Science*, pages 346–360. Springer Verlag, 2004.
7. H. Dicky, C. Dony, M. Huchard, and T. Libourel. On Automatic Class Insertion with Overloading. In *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'96*, pages 251–267, 1996.
8. J. Dvorak. Conceptual Entropy and Its Effect on Class Hierarchies. *IEEE Computer*, 27(6):59–63, 1994.
9. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, MA, 2002.
10. R. Godin and T.T. Chau. Comparaison d'algorithmes de construction de hiérarchies de classes. *L'Objet*, 5(3):321–338, 2000.
11. R. Godin, M. Huchard, C. Roume, and P. Valtchev. Inheritance And Automation: Where Are We Now? In *Object-Oriented Technology ECOOP Workshop Reader*, 2002.
12. R. Godin and H. Mili. Building and maintaining analysis-level class hierarchies using Galois lattices. In *Proceedings of OOPSLA'93, Washington (DC), USA*, pages 394–410, 1993.
13. R. Godin and H. Mili. Building and maintaining analysis-level class hierarchies using Galois lattices. In *Proceedings of OOPSLA'93, Washington (DC), USA*, special issue of ACM SIGPLAN Notices, 28(10), pages 394–410, 1993.
14. R. Godin, H. Mili, A. Arfi, G. W. Mineau, and R. Missaoui. A Tool for Building and Evaluating Class Hierarchies Based on a Concept Formation Approach. In *Proceedings of the OOPSLA 94 Workshop on Artificial Intelligence for Object-Oriented Software Engineering*, Portland, Oregon, 1994.
15. R. Godin, H. Mili, G. Mineau, R. Missaoui, A. Arfi, and T.T. Chau. Design of Class Hierarchies Based on Concept (Galois) Lattices. *Theory and Practice of Object Systems*, 4(2), 1998.

16. M. Huchard. Classification de classes contre classification d'instances. Evolution incrémentale dans les systèmes à objets basés sur des treillis de Galois. In *Actes de LMO'99: Langages et Modèles à Objets*, pages 179–196. Hermés, 1999.
17. M. Huchard, H. Dicky, and H. Leblanc. Galois lattice as a framework to specify algorithms building class hierarchies. *Theoretical Informatics and Applications*, 34:521–548, January 2000.
18. M. Huchard, M. Rouane Hacene, C. Roume, and P. Valtchev. Relational concept discovery in structured datasets. *Discrete Applied Mathematics*, submitted, 2004.
19. M. Huchard, C. Roume, and P. Valtchev. When concepts point at other concepts: the case of UML diagram reconstruction. In *Proceedings of the 2nd Workshop on Advances in Formal Concept Analysis for Knowledge Discovery in Databases (FCAKDD)*, pages 32–43, 2002.
20. R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, pages 22–35, June/July 1988.
21. T. Korson and J. D. McGregor. Technical Criteria for the Specification and Evaluation of Object-Oriented Libraries. *Software Engineering Journal*, 1992.
22. W.R. Lalonde. Designing families of data types using examplars. *ACM Transactions on Programming Languages and Systems*, 11(2):212–248, 1989.
23. H. Leblanc. *Sous-hiérarchies de Galois : un modèle pour la construction et l' evolution des hiérarchies d'objets (Galois sub-hierarchies: a model for construction and evolution of object hierarchies)*. PhD thesis, Université Montpellier 2, 2000.
24. K.J. Lieberherr, P. Bergstein, and I. Silva-Lepe. From Objects to Classes: Algorithms for Optimal Object-Oriented Design. *Journal of Software Engineering*, 6(4):205–228, 1991.
25. B. Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1988.
26. B. Meyer. *Conception et programmation par objets pour du logiciel de qualité*. Interеditions, Paris, 1990.
27. I. Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *Proceedings of OOPSLA'96, San Jose (CA), USA*, pages 235–250, 1996.
28. P. Rapicault and A. Napoli. Evolution d'une hiérarchie de classes par interclassement. *L'Objet*, 7(1-2), 2001.
29. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design*. Prentice Hall, 1991.
30. H. Sahraoui, R. Godin, and T. Miceli. Can Metrics Help to Bridge the Gap Between the Improvement of OO Design Quality and its Automations? In *Proceedings of the International Conference on Software Maintenance*, pages 154–162, 2000.
31. H.A. Sahraoui, H. Lounis, W. Melo, and H. Mili. A Concept Formation Based Approach to Object Identification in Procedural Code. *Automated Software Engineering*, 6:387–410, 1999.
32. G. Snelting and F. Tip. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems*, 22(3):540–582, May 2000.
33. G. Snelting and F. Tip. Semantics-based composition of class hierarchies. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, Malaga, Spain, June 2002.
34. P. Tonella. Concept analysis for module restructuring. *IEEE Transactions on Software Engineering*, 27(4):351–363, 2001.
35. P. Valtchev, M. Hacene Rouane, M. Huchard, and C. Roume. Extracting Formal Concepts out of Relational Data. In E. SanJuan, A. Berry, A. Sigayret, and A. Napoli, editors, *Proceedings of the 4th Intl. Conference Journées de l'Informatique Messine (JIM'03): Knowledge Discovery and Discrete Mathematics, Metz (FR), 3-6 September*, pages 37–49. INRIA, 2003.
36. A. Yahia, L. Lakhal, R. Cicchetti, and J.P. Bordat. iO2 - An Algorithmic Method for Building Inheritance Graphs in Object Database Design. In *Proceedings of the 15th International Conference on Conceptual Modeling ER'96*, volume 1157, pages 422–437, 1996.