A Generic Approach for Class Model Normalization

Jean-Rémy Falleri, Marianne Huchard, Clémentine Nebut

A generic approach for class model normalization *

Jean-Rémy Falleri Marianne Huchard Clémentine Nebut
LIRMM, CNRS and Université Montpellier 2,
161, rue Ada, 34392 Montpellier cedex 5, France
E-mail: {falleri, huchard, nebut}@lirmm.fr

Abstract

Designing and maintaining a huge class model is a very complex task. When an object oriented software or model becomes bigger and bigger, duplicated elements start to appear, decreasing the readability and the maintainability of the software. In this paper, we present an approach, implemented in a tool and validated by a case study, that helps software architects designing and improving their class models. Since many different languages (UML, EMOF, Java, ...) allow to express class models, this approach has been made generic i.e. capable of dealing with any language described by a meta-model. Using this approach, software architects will be able to design and maintain more efficiently their class models.

1. Introduction

Designing and maintaining class models is a crucial task in object oriented software design. A well designed class model makes the software easier to understand, maintain and reuse. Designing a class model is an iterative task: classes, properties/attributes, associations and methods are added and modified as the software evolves.

When the software reaches a large size (in terms of number of classes, methods, attributes and associations), it is almost impossible for a software architect to know each detail of the architecture. Consequently, element duplications are unintentionally introduced. For instance, many class models contain two different classes that own a property named `name` or `ident`. Because of the huge number of methods and properties, removing code duplications becomes a very difficult task and can hardly be manually achieved. Table 1 gives an insight of the number of duplicated attributes names (identifiers) in four class models that will be used in the case study (see section 5). UML2 and Docbook are two metamodels designed with Ecore, Apache Common Collections (ACC) and Minjava are written in Java. Those name duplications do not necessarily imply redundant declarations since two attributes can have the same name and different meanings. However, it gives an indication on the actual number of duplicated attributes.

	Docbook	UML2	Minjava	ACC
#Classes	40	246	29	250
#Attributes	183	615	340	544
#Attrib. name duplications	161	319	63	373

For a given identifier `I` duplicated `n` times, we count `n` duplications (and not one).

Table 1. Attribute name (identifier) duplications in four class models

Literature on class model design shows that it is possible to compute normal forms of a given class model. In [18], five different normal forms are defined: two normal forms regarding the classes and their attributes, two for classes and their methods and the last one concerning classes and associations. All these normal forms guarantee that a given attribute or method will appear exactly once in the whole class model and that inheritance links correspond to attribute and method sets inclusion or refinement. These forms differ by the amount of multiple inheritance links used to address the previous criterion (for instance, classes that do not introduce an attribute or a method can be removed and replaced by multiple inheritance links). In the rest of the paper, a class model is said to be in normal form if and only if there is no redundancies in it and inheritance links correspond to attribute and method sets inclusion or refinement.

The contribution presented in this paper is a generic approach and tool using Model-Driven Engineering and Relational Concept Analysis to perform class model normalization, and a case study on real world class models coming from different languages (Java and Ecore).

Formal Concept Analysis [16] has proven [18] to be an

*France Télécom R&D has supported this work (CPRE 5326)

efficient approach to perform class model normalization. FCA is a clustering method that automatically classifies elements described by binary attributes. When applied to class models, it can find duplicated properties or methods and produce a normalized class model that minimizes the number of classes needed to introduce all the properties and methods of the initial model. On the other hand, FCA is unable to deal with other kind of specialization/generalization mechanisms existing in class models. For instance a method that redefines a method is not going to be created by a FCA process.

Relational Concept Analysis [6], an extension of Formal Concept Analysis, is an even more efficient approach to deal with object oriented softwares. It is a relational extension of FCA that allows to deal with entities described by binary attributes and by relations with the other entities. Using RCA, most of the constructions (invariant or covariant method redefinition, covariant attribute redefinition) proposed by object oriented languages are supported. More details about FCA and RCA are presented in Section 2.

Class models can be expressed in many different languages, such as modeling languages (UML, Ecore, ...) or object-oriented programming languages (Java, C++, ...). The crucial point tackled by our approach is the difficulty to deal with all these different languages : before applying an FCA or RCA process, the source data have to be translated into the input FCA data format. We propose to use Model Driven Engineering to deal with this issue.

The rest of this paper is structured as follows. Section 2 gives a quick presentation of formal and relational concept analysis. Section 3 shows how we use Model Driven Engineering to build a generic approach and tool, that is presented in Section 4. Section 5 presents and discusses a case study of the application of our tool to real world models and softwares. Section 6 discusses related world and concludes.

2. Formal and Relational Concept Analysis

Formal Concept Analysis [19] is a clustering method that classifies a set of entities described by binary attributes. More formally, let $K = (E, A, R)$ be a formal context. E is a set of entities, A is a set of attributes and R a binary relation such as $R \subseteq E \times A$. A sample formal context is shown at the right of Figure 1. In this context, entities are the rows and attributes are the columns.

With a formal context, several concepts can be produced. A concept is a set of entities that share several attributes. It can be considered as an abstraction of these entities. More formally, a concept is a pair (X, Y) with $X \subseteq E$, $Y \subseteq A$ and $X = \{e \in E | \forall y \in Y, (e, y) \in R\}$ is the extent (covered entities), $Y = \{a \in A | \forall x \in X, (x, a) \in R\}$ is the intent (shared attributes).

As the definition states, the sets of entities and attributes are maximal, i.e. there is no other entity that belongs to the concept extent and owns all the attributes of the intent. Moreover, there is no other attribute that belongs to the concept intent and that is owned by all the entities of the extent. These properties ensure maximal factorization of attributes, and in the context of class model, avoids property and method duplications.

The concepts can be organized in a specialization lattice: a concept c_1 is lower than a concept c_2 if the extent of c_1 is included in the extent of c_2 (and inversely, the intent of c_2 is included in the intent of c_1). The specialization lattice ensures, in the context of class model normalization, that inheritance or specialization links respect property/method sets inclusion and refinement. A sample lattice corresponding to the context of Figure 1 is shown at the left of Figure 2.

Three steps are required to apply formal concept analysis on a class model. First, the class model is converted into a formal context. This step is shown on Figure 1. Each class of the class model is converted into an entity in the formal context. The properties of the class are converted into attributes in the formal context, and the binary relation of the formal context is built according to attribute possession.

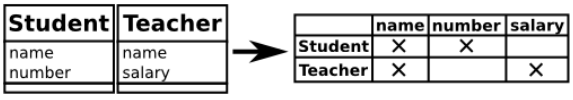

Figure 1. First step of FCA on a UML model

Second, a concept lattice is built, according to the formal context. This concept lattice will contain concepts that represent the existing entities (and thus the classes) of the formal context, and new concepts that will lead to the creation of new classes. The last step is to build a class model according to the concept lattice. This step is shown in Figure 2. It is clear that the output class model is normalized whereas the input class model was not. This normal form is called *attribute lattice factored form* in [18].

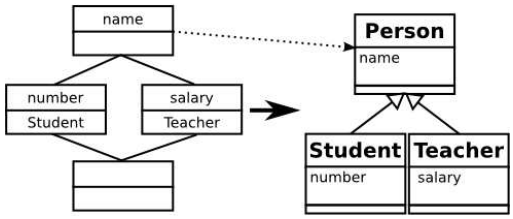

Figure 2. Second step of FCA on a UML model

Formal Concept Analysis is powerful to distribute attributes in a class hierarchy, but is unable to deal with relational descriptions. As an example, let us consider the

class model in the top of Figure 3. The same conversion and application of FCA on this model, as previously described, would lead to the creation of the model shown in the right of Figure 3. The resulting model, even if it is in normal form, could still be improved. A new attribute with type Person could be introduced in the class Person. Then, the *friends* and *colleagues* properties should redefine this new attribute.

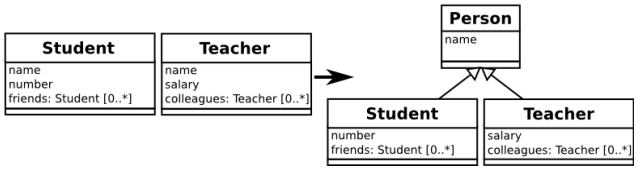

Figure 3. Limitations of FCA on a UML model

Relational Concept Analysis [6, 19] is an extension of FCA. It is designed to take into account entities described by binary attributes with relations linking them. In RCA, instead of having just one formal context, there is one formal context for each kind of entities. Then these formal contexts are filled out with other contexts that show relations between entities coming from one context and entities coming from another context (which can be the same). More formally, a Relational Context Family (RCF) is a pair $F = (K, L)$ where K is a set of formal contexts where $K_i = (E_i, A_i, R_i)$ and L a set of relational contexts, $L_i = (E_a, E_b, R_i)$ with $R_i \subseteq E_a \times E_b$. Figure 4 shows the relational context family corresponding to the class model at the left of Figure 3.

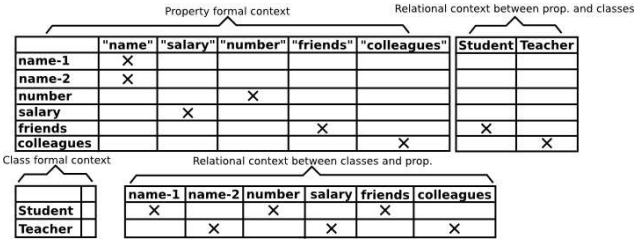

Figure 4. Applying RCA on a UML model: produced contexts

An iterative lattice construction is applied on the relational context family. A concept lattice is built for each formal context K_i of the Relational Context Family. The discovered concepts of these lattices are injected as new entities in the RCF, and new lattices are built. This iterative construction stops whenever for each category of entities, the lattices built while performing two successive steps are isomorphic. The set of lattices produced after each step of the process is called a Concept Lattice Family (CLF). The class model in Figure 5 has been produced from the contexts of Figure 4.

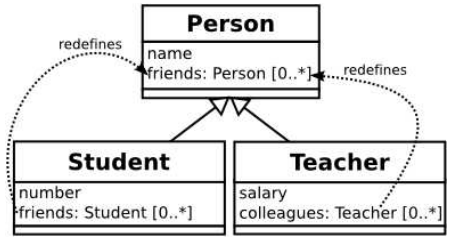

Figure 5. RCA result on a UML model

The description of FCA and RCA given is this section are very brief, but more detailed explanations, with examples, can be found in [18, 2, 14].

3. Generic encoding of the input and the output : issues and MDE-based solution

In this section we point out the problems emerging when building a single RCA tool handling different input languages, and explain why MDE solves the problems.

When applying a formal method on software artifacts, a problem often emerges: software artifacts have to be translated into a format from which it is possible to apply the formal method. With RCA for instance, the class models have to be encoded into formal contexts. To build an FCA- or RCA-based tool able to deal with a large range of input data formats, it is often necessary to develop as well a great number of encoders (see Figure 6). To apply RCA on Java programs, it is necessary to dispose of a Java grammar and parser, and to implement a program that encodes Java abstract syntactic trees into formal contexts. To apply RCA on a UML model, an XML or XMI parser is required, and again, a translator between XMI and formal contexts has to be created.

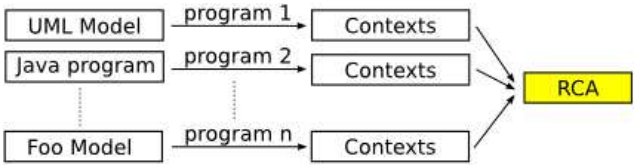

Figure 6. A naive tool approach

It is easy to see that to deal with n different languages, $n+1$ programs are required (the last one to apply the formal method). Worse, if we consider that the output of the formal method has to be converted back into the initial format, like in RCA, $2n + 1$ programs are required. For technological reasons, these programs are likely not to be coded with the

same language (because of parsers availability). This architecture raises many problems. If the formal method input data format is modified, n programs have to be modified. If a different configuration of the translation has to be tested, the n programs have to be modified again. Therefore, this architecture requires too much coding effort to cope with a large range of input data. The solution we propose to tackle this issue is based on Model-Driven Engineering.

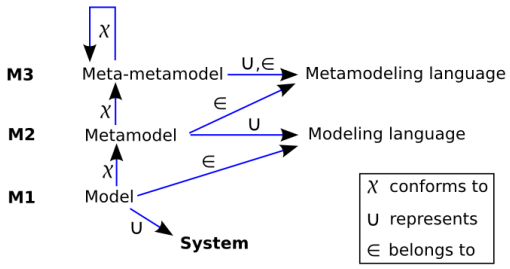

Figure 7. The meta-modeling hierarchy

Model Driven Engineering [21] is a recent software development paradigm. It was introduced to deal more with abstractions rather than code. In a MDE-based development, every produced or used artifact (including code) is a model, whose structure is defined by a meta-model (a model is said to conform to a meta-model). To pragmatically handle two models that conform to two different meta-models (for example to transform a UML model into a Relational Database model), a program has to be written, dealing with both meta-models. For that purpose, MDE assumes the existence of a unique meta-metamodel. Such a meta-metamodel allows to define how a meta-model is structured. Mainly, two meta-metamodels are used: EMOF [24] (defined by the OMG) and Ecore [11] (defined by Eclipse). Since we have built our tool with the Eclipse platform, we have chosen the meta-metamodel Ecore. However, Ecore and EMOF have no significant differences. In the following, we will use Ecore. The meta-metamodel is the last level in the modeling hierarchy (shown in Figure 7), and is expressive enough to describe itself. Therefore a meta-meta-metamodel is not necessary.

In the introduction, we explained that the goal of this paper is to provide a generic approach with a tool allowing to deal with several different languages. Such a tool must be able to translate class models (either UML or Java models for example) to RCA contexts. Yet, Ecore can describe either a UML model or Java code. Therefore, such a translator can be considered as a model transformation taking as input a model described by a meta-model written with Ecore, and producing as output RCA contexts. The opposite translator, that takes as input RCA lattices and produces a model in the same format as the inital one, is also a model transformation.

All the information from a UML model or Java code is not relevant for an RCA process. For example, the fact that two classes are abstract does not imply to build an abstraction of those two classes: this fact has not to be taken as a reason to build abstractions, and thus has not to be encoded. The translators thus need to know what parts of the models have to be encoded for the RCA process, and also the specialization links that can exist in the model (method redefinition or class inheritance for example). We propose to give the translators this configuration information by the way of a configuration model, specifying all this information in terms of elements of the input or output metamodel.

The next section details our MDE-based approach and tool, that allows an easy definition of RCA translators.

4. Generic class model normalization

In this section, we describe our approach, summurized in Figure 8, that integrates RCA and MDE to perform class model normalization. Three successive model transformations are defined:

1. *encoding*: transforms the input class model (which can be Java code, a UML class model, ...) into a Relational Context Family representing this class model,
2. *RCA*: apply the RCA process on the previously generated RCF to build a Concept Lattice Family,
3. *decoding*: transforms the previously built CLF into a class model conform to the same meta-model as the input model.

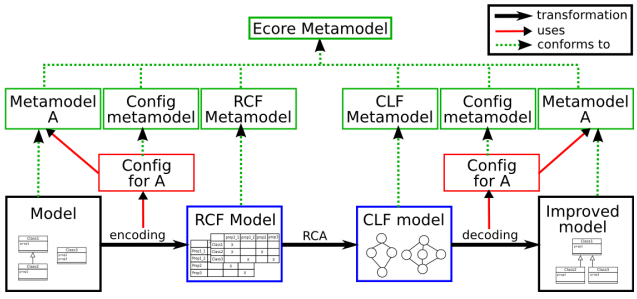

Figure 8. Process overview

In our process, the first (encoding) and the third (decoding) transformations have to be generic. By generic, we mean that they are written independently from the meta-model to which the input/output models conform. Yet, a Java class model cannot be transformed into a RCF like a UML model, at least because names of meta-classes and meta-references in both underlying metamodels are not the same. Since it is not possible to automatically detect in

a meta-model the elements that are interesting and worth injecting in the RCA process, configuration data have to be furnished to the *encoding* and *decoding* transformations. This configuration is based on the meta-model of the class model to analyse or to produce, and dynamically tunes the behaviour of the generic transformations.

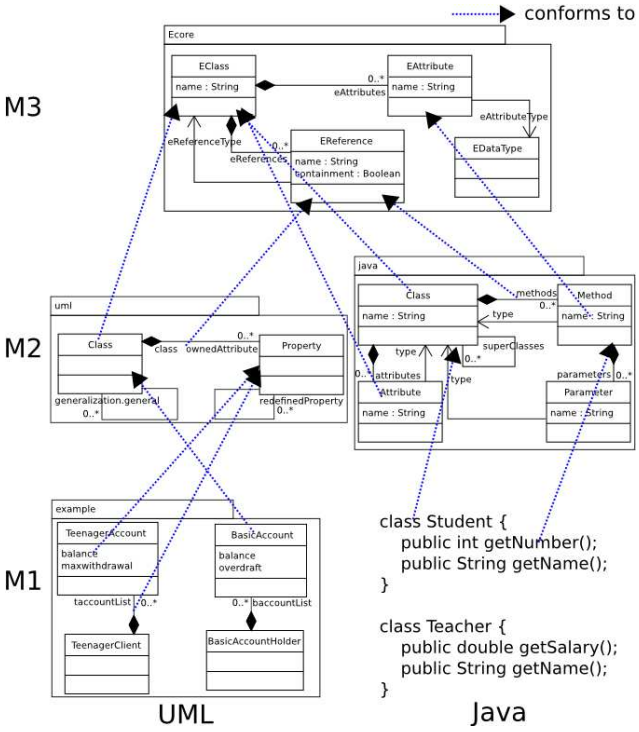

Figure 9. The sample models

We use two sample models to show how the *encoding* transformation works. The first one (at the bottom and lhs of Figure 9) is a simple UML model, the second one (at the bottom and rhs of Figure 9) is a tiny piece of Java code. Figure 9 places the two examples in the meta-modeling hierarchy: meta-models of the models are represented, and the meta-metamodel (Ecore in this example) of the meta-models as well. The Java, UML and Ecore meta-models are presented in a reduced form that only contains information relevant to illustrate our process. The dotted arrows show the *conforms to* links between elements and meta-elements. Information that will tune the *encoding* and *decoding* transformations will come from the *M2* level. In the rest of the section, we will show how we use those models and meta-models through the *encoding* transformation.

Let us suppose that we want to apply the same RCA configuration as in Figure 4 to the sample UML model. To do that, we want to create two formal contexts, one describing the classes and one describing the properties. In order to merge properties, the name of the properties has to be used as an attribute in the properties context. Two relational contexts are also required: one describing the *ownedAttribute* relation between classes and properties, and one describing the *type* relation between the properties and the classes. For the Java model, the configuration has to be different. Indeed, the Java model does not contains properties, only methods. Therefore we want to merge methods based on their names to create super-classes. So for this example, we will have two formal contexts: one for the classes and the other for the methods. The methods context will use the name of the methods as an attribute, and a relational context will describe the *methods* relation between classes and methods (methods introduced by a class). In order to give that kind of information to the *encoding* and *decoding* transformations, we have introduced a configuration meta-model, shown in Figure 11.

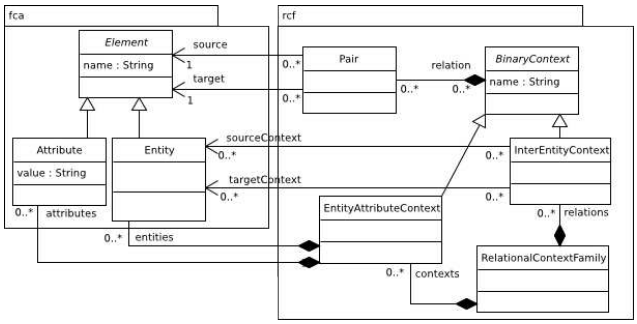

Figure 10. The Relational Context Family metamodel

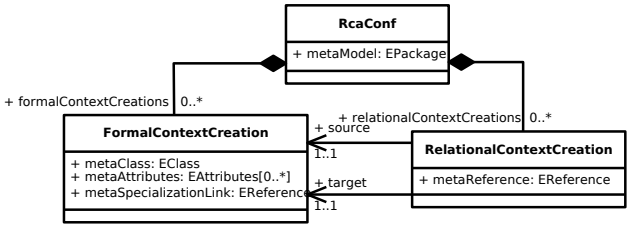

Figure 11. The encoding/decoding configuration metamodel

The *encoding* transformation uses two models to fulfill its goal: a class model (UML, Java, ...) and a configuration model conform to the configuration meta-model previously shown. To remain in the MDE paradigm, we created a meta-model for the RCF that will be produced by this transformation (Figure 10). This transformation works as follows. First, a formal context is created for each *FormalContextCreation* element in the configuration model. Entities of this formal context are the elements coming from the class model which are conform to the meta-class defined in

the *metaClass* attribute of the *FormalContextCreation* element. The attributes of this formal context will be created according to the values of the *metaAttributes* attribute of the *FormalContextCreation* element.

Figure 12 shows in a textual format the configuration model used to encode the sample UML model. According to this configuration model, two formal contexts will be created: one for the classes (*MetaClass class*) and one for the properties (*MetaClass Property*). No attributes will be created in the classes formal context. The value of the *name* attribute from the properties will be used in the properties formal context. Figure 13 shows the two formal contexts $K_{property}$ and K_{class} created using both the sample UML model and the sample configuration model.

```
RCA Config. for UML Class Models:

Formal Context Creations:
- MetaClass Class: metaAttributes = [],
   metaSpecializationLink = "generalization.general"
- MetaClass Property: metaAttributes = ["name"],
   metaSpecializationLink = "redefinedProperty"

Relational Context Creations:
- MetaReference ownedAttribute: source = Class,
   target = Property
- MetaReference type: source = Property,
   target = Class
```

Figure 12. UML configuration model

Kproperty

	name: 'balance'	name: 'maxwithdrawal'	name: 'overdraft'	name: 'baccountlist'	name: 'taccountlist'
balance_1	X				
balance_2	X				
maxwithdrawal		X			
baccountlist				X	
taccountlist					X
overdraft			X		

Kclass

TeenagerAccount
BasicAccount
BasicAccountHolder
TeenagerClient

RownedAttribute

	balance_1	balance_2	maxwithdrawal	baccountlist	taccountlist	overdraft
TeenagerAccount	X		X			
BasicAccount		X				X
BasicAccountHolder				X		
TeenagerClient					X	

Rtype

	TeenagerAccount	BasicAccount	BasicAccountHolder	TeenagerClient
balance_1				
balance_2				
maxwithdrawal				
baccountlist		X		
taccountlist	X			
overdraft				

Figure 13. The generated UML contexts

After having created the formal contexts, the *encoding* transformation creates the relational contexts. One relational context will be created for each *RelationalContextCreation* element from the configuration model. The *source* and *target* attributes from the *RelationalContextCreation* element will define which are the entities involved in this relational context. The source entities are the entities of the *FormalContextCreation* defined as source of the *RelationalContextCreation* element, and so on for the target entities. Then, for each source entity, the *encoding* transformation will search if relations with the target entities of the type defined in the *metaReference* attribute of the *RelationContextCreation* element exist in the input class model. Those relations will be reported into the relational context. In the UML configuration model of Figure 12, we can see that two relational contexts will be created (they are shown in Figure 13):

- $R_{ownedAttribute}$ stems from the *MetaReference ownedAttribute* in the configuration model. It links the the classes and the properties: a pair will be added in the relation each time a class owns an attribute.
- R_{type} stems from the *MetaReference type* in the configuration model. It links the properties and the classes: a pair will be added in the relation each time a property is typed by a class.

Similarly, Figure 14 shows the configuration model used to apply the RCA process on the sample Java code. Figure 15 shows the context produced by the *encoding* transformation, using the sample Java code and the sample Java configuration model.

```
RCA Config. for Java Class Models:

Formal Context Creations:
- MetaClass Class: metaAttributes = [],
   metaSpecializationLink = "generalization.general"
- MetaClass Method: metaAttributes = ["name"]

Relational Context Creations:
- MetaReference methods: source = Class, target = Method
```

Figure 14. Java configuration model

Kclass

Student
Teacher

Kmethod

	'getNumber'	'getName'	'getSalary'
getNumber	X		
getName_1		X	
getName_2		X	
getSalary			

Rmethods

	getNumber	getName_1	getName_2	getSalary
Student	X	X		
Teacher			X	X

Figure 15. The generated Java contexts

Since we have not detailed how is structured a Concept Lattice Family in Section 2, we will not explain in details

how works the *decoding* transformation. But the principle of this transformations is the same as the one of the *encoding* transformation, and the same configuration model is used to perform this transformation.

The whole RCA process is very complex, and involves lots of details that have not been introduced here for the sake of clarity. Having implemented the whole process with model transformations allowed us to clearly identify the variation points in the involved algorithms and add them in the configuration metamodel. In that way, a RCA expert can easily fine-tune the process, without modifying a single line of code of the tool.

5. Case study

To evaluate our class model normalization approach, we carried out an experiment on four actual class models. Two of them, UML [12] and Docbook [27], are design models written in Ecore. The two others, Apache Commons Collections (ACC) [15] and Minjava [13], are implementation models, obtained by reverse-engineering on Java code. UML stands for the UML 2.0 meta-model. Docbook is a meta-model of the Docbook language. Apache Commons Collections is a Java library that extends the Java collections. Minjava is a Java reverse engineering tool that analyses Java byte-code and produces an Ecore compliant Java model conform to a simple Java meta-model. This tool can restrict the extraction of Java code to some packages. When building our sample models, we chose to restrict the extraction of Java entities to the program itself, and blocked the extraction of the Java standard library (except base types). So when a Java class introduces an attribute typed by a class included in the standard Java API (for instance a *LinkedList*), the attribute appears as not typed in the resulting model. We tested three different configurations of our approach on these models:

1. Basic FCA configuration (*FCA1*): it corresponds to the one in [18], that generates a class and a property context and analyses the attribute possession to discover super-classes, based on attribute names. Figure 16 shows this configuration applied to the Ecore meta-model.

2. Enhanced FCA configuration (*FCA2*): same as the previous configuration, but using information specific to the input language (static keyword in Java, cardinality in Ecore) to avoid incorrect generalizations. Figure 17 shows this configuration applied to the Ecore meta-model.

3. Enhanced Properties configuration (*RCA*): a RCA configuration that generates a class and a property context and analyses the attribute possession and type to discover super-classes and redefined properties. Figure 18 shows this configuration applied to the Ecore meta-model.

Since we used two different kinds of models (Ecore and Java) in our experiments, these three configurations has been defined for the two languages. Therefore, six configuration models have been designed.

```
Basic FCA Config for Ecore

Formal Context Creations:
- MetaClass EClass: metaAttributes = [],
   metaSpecializationLink = "generalization.general"
- MetaClass EAttribute: metaAttributes = ["name"]
- MetaClass EReference: metaAttributes = ["name"]

Relational Context Creations:
- MetaReference eStructuralFeatures: source = EClass,
    target = EAttribute
- MetaReference eStructuralFeatures: source = EClass,
    target = EReference
```

Figure 16. FCA1 configuration for Ecore

```
Enhanced FCA Config for Ecore

Formal Context Creations:
- MetaClass EClass: metaAttributes = [],
   metaSpecializationLink = "generalization.general"
- MetaClass EAttribute: metaAttributes = ["name",
    "upperBound","lowerBound","derived"]
- MetaClass EReference: metaAttributes = ["name",
    "upperBound","lowerBound","derived"]

Relational Context Creations:
- MetaReference eStructuralFeatures: source = EClass,
    target = EAttribute
- MetaReference eStructuralFeatures: source = EClass,
    target = EReference
```

Figure 17. FCA2 configuration for Ecore

```
Enhanced RCA Properties Config for Ecore

Formal Context Creations:
- MetaClass EClass: metaAttributes = [],
   metaSpecializationLink = "generalization.general"
- MetaClass EAttribute: metaAttributes = ["name",
    "upperBound","lowerBound","derived"]
- MetaClass EReference: metaAttributes = ["name",
    "upperBound","lowerBound","derived"]

Relational Context Creations:
- MetaReference eStructuralFeatures: source = EClass,
    target = EAttribute
- MetaReference eStructuralFeatures: source = EClass,
    target = EReference
- MetaReference eType: source=EReference, target=EClass
```

Figure 18. RCA configuration for Ecore

To present the results of the application of RCA to our sample models, we use the produced Concept Lattices Family. We classify the concepts of these lattices into three disjoint categories:

- $ExistingConcepts$: these concepts represent elements that were already present in the input class model,
- $NewConcepts$: these concepts represent elements created during the RCA process,
- $MergeConcepts$: these concepts represent the merge of existing elements from the input model.

The $ExistingConcepts$ set is not really interesting since it contains only concepts representing the input entities. The $NewConcepts$ set is very interesting. It contains the concepts that may introduce new useful elements (abstractions of existing ones) in the class model. The $MergeConcepts$ set is also interesting, since it contains the elements from the source model that have been considered as similar and therefore have led to the creation of the new elements. To present the result of our case study, we choose to use the two following quantities:

- N, the number of new elements *i.e.* $|NewConcepts|$,
- M, the number of merges *i.e.* $|MergeConcepts|$.

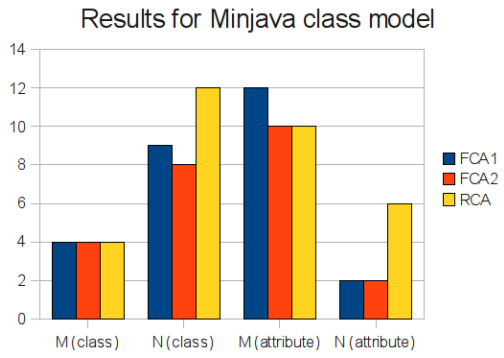

Figure 19. Results for Minjava class model

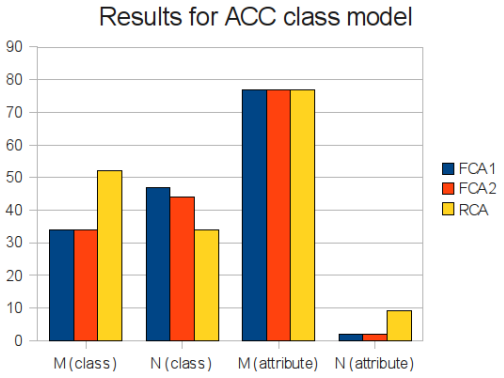

Figure 20. Results for ACC class model

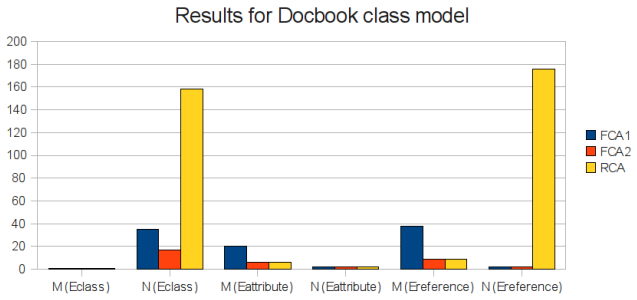

Figure 21. Results for Docbook class model

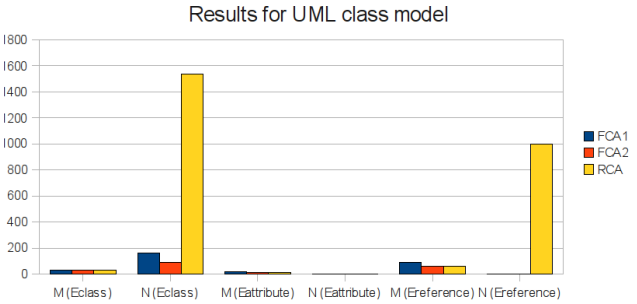

Figure 22. Results for UML class model

Figures 19-22 show the results of the application of the different RCA processes to our sample class models. It is clear the FCA1 configuration produces more merge than the two other configurations. This is the expected result because the two other configurations use information specific to the class model language in order to avoid incorrect merges. The RCA configuration is the one that produces the greatest number of new elements. This is also the expected result, because it uses the type of the properties to create more abstract new properties that led to the creation of more new super-classes.

The previous results show how the different configurations of the RCA process behave, but are unable to show the quality of these results. Metrics are a way of assessing quality, but they are not so easy to use: based on current inheritance metrics from [5, 23], it has been shown in [10] that inheritance metrics (associated with size metrics) are useful in measuring software stability, but don't really help in detecting concrete design problems.

In [17], the case study uses a structural metric to analyze the result of FCA application on real world class hierarchies. The chosen metric, called *M2* is derived of the *M1* metric introduced in [22]. This metric measures redundancy and inheritance quality. Basically, *M2* is a weighted sum of the number of attributes and the number of inheritance links. To defavor the use of multiple inheritance, for a given class the inheritance links count as double after the first one. The lower metric *M2* is, the better the class model

is designed.

Unfortunately, this metric can lead to wrong analysis of the class model. If we imagine an output class model where a super-class has been found but is not correct (for instance because of homographs), the *M2* metric will still consider this output model as better than the input one. Moreover, this metric is not compatible with the use of redefined properties or methods. If we use the class model shown in Figure 3, the metric *M2* will be 24 for the input model, 22 for the output model without attribute redefinition and 26 for the output class model with attribute redefinition. This clearly shows that this metric is unable to correctly measure the quality of a class model design when attribute redefinition is used.

Two research directions are now open. Firstly, results from FCA/RCA on class model can be assessed using recent proposals and results on specialization quality measurement [9, 4]. Secondly, specific metrics can be introduced for FCA and RCA based on human judgement. To compute our metrics, a human analysis of the concepts of the $NewConcepts$ and the $MergeConcepts$ sets is required. An architect, preferably with a good knowledge of the input class model, counts the number of concepts in these two sets that are considered as correct. Our four metrics are:

- cn: number of concepts included in the *NewConcepts* set that are considered as correct; a rate is obtained with $cnr = cn/|NewConcepts|$;
- cm: number of concepts included in the *MergeConcepts* set that are considered as correct; a rate is obtained with $cmr = cm/|MergeConcepts|$.

Results of these metrics on Minjava are shown in Tables 2 and 3. Concept correction has been assessed by the designer of Minjava. These results confirm what was expected from the quantitative results. *FCA1* is the configuration that produces the most of incorrect merges and *RCA* produces the most incorrect new concepts. On the other hand, new concepts produced by *RCA* could not have been created using a FCA configuration and can contains useful concepts. However it is necessary to find a way to analyse those new concepts in a semi automatic way, because they are too numerous to be analysed by hand.

6. Related work and conclusion

Our previous works ([20], [7], [2]) were the origins of the presented generic approach in model transformations using RCA. [8] showed an implementation using Objecteering and Galicia [28], and there also exists an implementation using Galicia with specific modules translating XMI files into RCF, and CLF into XMI files. In [2], we have shown our approach specifically using UML metamodel.

	FCA1	FCA2	RCA
$\|MergeConcepts\|$	12	10	10
cm	10	10	10
cmr	0.83	1	1
$\|NewConcepts\|$	2	2	6
cn	0	0	0
cnr	0	0	0

Table 2. Results on attributes for Minjava

	FCA1	FCA2	RCA
$\|MergeConcepts\|$	4	4	4
cm	3	3	3
cmr	0.75	0.75	0.75
$\|NewConcepts\|$	9	8	12
cn	5	5	5
cnr	0.56	0.63	0.42

Table 3. Results on classes for Minjava

In all the approaches, the implementation of RCA-building algorithms were crucial in the computation time. To our knowledge, these were the only RCA-based approaches in the context of Model Driven Engineering. Related to model refactoring, the majority of the contributions on refactoring addresses the code level, but the recent interest for model-driven approaches led to several works on model refactoring, in particular UML refactoring [25]. Most of the research focuses on small and atomic model transformations (adding a class, adding an association), except the community working on design pattern application by model refactoring (for example [26]). Our main claim is to show the evolution of our approach to be generic and independent of the implied models. Works, such as shown in [3], confirm our hypothesis about the degree of complexity that implies the model transformations.

We have presented in this paper a theory and a tool allowing to normalize class models based on different metamodels. The normalization process is based on Relational Concept Analysis. A case study has been conducted to demonstrate that the FCA and RCA process can be adapted just modifying the configuration model of the underlying model transformations. A quantitative analysis has been given in terms of dedicated metrics on the obtained results. The experiments conducted with the tool confirmed us in the intuitive idea that some FCA and RCA configurations allow to discover lots of abstractions, among them a small number of very relevant ones (that cannot be found with simpler configurations), and a large number of a poorly-interesting ones. We thus plan to work in two main directions. First, we plan to make a large qualitative analysis with domain

experts on several case studies, to determine if a trade-off can be found in the configurations to detect a maximum number of relevant abstractions, with a minimum number of low value-added generated abstractions. Second, we will continue current work on natural language analysis based on lexical nets to detect on the fly or afterward the interesting abstractions, to name discovered abstractions, and to measure the relevancy of an abstraction.

References

[1] *Formal Concept Analysis, Foundations and Applications*, volume 3626 of *Lecture Notes in Computer Science*. Springer, 2005.

[2] G. Arévalo, J.-R. Falleri, M. Huchard, and C. Nebut. Building abstractions in class models: Formal concept analysis in a model-driven approach. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 513–527. Springer, 2006.

[3] J. Bézivin, B. Rumpe, A. Schuerr, and L. Tratt. Model transformations in practice workshop. In J.-M. Bruel, editor, *MoDELS 2005 Workshops*, pages 120–127. Springer-Verlag Berlin Heidelberg, 2006.

[4] K. M. Breesam. Metrics for object-oriented design focusing on class inheritance metrics. In *DepCoS-RELCOMEX*, pages 231–237. IEEE Computer Society, 2007.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.

[6] M. Dao, M. Huchard, M. R. Hacene, C. Roume, and P. Valtchev. Improving generalization level in uml models iterative cross generalization in practice. In Wolff et al. [29], pages 346–360.

[7] M. Dao, M. Huchard, M. R. Hacene, C. Roume, and P. Valtchev. Improving Generalization Level in UML Models Iterative Cross Generalization in Practice. In Wolff et al. [29].

[8] M. Dao, M. Huchard, M. R. Hacène, C. Roume, and P. Valtchev. Towards practical tools for mining abstractions in uml models. In *Proc. of the ICEIS 2006 conference*, pages 276–283, 2006.

[9] M. Dao, M. Huchard, T. Libourel, C. Roume, and H. Leblanc. A new approach to factorization - introducing metrics. In *IEEE METRICS*, pages 227–236. IEEE Computer Society, 2002.

[10] S. Demeyer and S. Ducasse. Metrics, do they really help? In J. Malenfant and R. Rousseau, editors, *LMO*, pages 69–82. Hermès, 1999.

[11] Eclipse. The Eclipse Modeling Framework. http://www.eclipse.org/emf, 2005.

[12] Eclipse. UML2 EMF Plugin. http://www.eclipse.org/modeling/mdt/?project=uml2, 2008.

[13] J.-R. Falleri. Minjava. http://code.google.com/p/minjava/, 2008.

[14] J.-R. Falleri, M. Huchard, C. Nebut, and G. Arévalo. A model driven engineering approach for making generic fca/rca tools. In J. Diatta, P. Eklund, and M. Liquière, editors, *Proceedings of the Fifth International Conference on Concept Lattices and Their Applications (CLA'07)*, pages 229–252, 2007.

[15] A. Foundation. Apache Commons Collections. http://commons.apache.org/collections, 2008.

[16] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1997.

[17] R. Godin, H. Mili, G. Mineau, R. Missaoui, A. Arfi, and T. Chau. Design of class hierarchies based on concept,(Galois) lattices. *Theory and Practice of Object Systems*, 4(2):117–134, 1998.

[18] R. Godin and P. Valtchev. Formal concept analysis-based class hierarchy design in object-oriented software development. In *Formal Concept Analysis* [1], pages 304–323.

[19] M. Huchard, M. R. Hacene, C. Roume, and P. Valtchev. Relational concept discovery in structured datasets. *Ann. Math. Artif. Intell.*, 49(1-4):39–76, 2007.

[20] M. Huchard, C. Roume, and P. Valtchev. When concepts point at other concepts: the case of UML diagram reconstruction. In *FCAKDD 2002, Advances in Formal Concept Analysis for Knowledge Discovery in Databases, Int. workshop ECAI 2002*, pages 32–43, Lyon, juillet 2002.

[21] S. Kent. Model Driven Engineering. In M. J. Butler, L. Petre, and K. Sere, editors, *IFM*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer, 2002.

[22] K. Lieberherr, P. Bergstein, and I. Silva-Lepe. From objects to classes: algorithms for optimal object-orienteddesign. *Software Engineering Journal*, 6(4):205–228, 1991.

[23] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[24] OMG. MOF 2.0 core specification. http://www.omg.org/cgi-bin/doc?ptc/2004-10-15, 2004.

[25] G. Sunyé, D. Pollet, Y. Le Traon, and J.-M. Jézéquel. Refactoring UML models. In *Proc. Unified Modeling Language Conf.*, 2001.

[26] L. Tokuda and D. Batory. Automated software evolution via design pattern transformations. In *Proc. of the Int'l Symp. on Applied Corporate Computing*, 1995.

[27] Triskell. Docbook metamodel. http://www.kermeta.org, 2008.

[28] P. Valtchev, D. Grosser, C. Roume, and M. R. Hacene. GALICIA: an open platform for lattices. In A. d. M. B. Ganter, editor, *Using Conceptual Structures: Contributions to ICCS'03*, pages 241–254, Aachen (DE), 2003. Shaker Verlag.

[29] K. E. Wolff, H. D. Pfeiffer, and H. S. Delugach, editors. *Conceptual Structures at Work: 12th International Conference on Conceptual Structures, ICCS 2004, Huntsville, AL, USA, July 19-23, 2004. Proceedings*, volume 3127 of *Lecture Notes in Computer Science*. Springer, 2004.