Use of Model Driven Engineering in Building Generic FCA/RCA Tools *

J.-R. Falleri[1], G. Arévalo[2], M. Huchard[1], and C. Nebut[1]

[1] LIRMM, CNRS and Université de Montpellier 2,
161, rue Ada, 34392 Montpellier cedex 5, France
{falleri, huchard, nebut}@lirmm.fr
[2] LIFIA – Facultad de Informatica (UNLP)
La Plata, Argentina
garevalo@sol.info.unlp.edu.ar

Abstract. Within maintenance software methodologies that analyze existing applications, Relational Concept Analysis (RCA) is an efficient approach to build abstractions in any language, using the existing relations between different software artifacts. Nowadays, there are several RCA-based tools, where a critical aspect is the lack of genericity in the FCA mapping to translate input (and output) data to (and from) formal/relational contexts. Most of the tools provide specific translators that can be used only with the analyzed application domain and their code needs to be changed when the framework of analysis evolves. Using Model-Driven Engineering, we propose a generic encoding/decoding process, which performs this translation with only the configuration of an encoder/decoder tool. This approach eases the integration of a FCA/RCA process in a tool and facilitates its usage on a wide range of input data formats.

1 Introduction

Within maintenance software methodologies that analyze existing applications, Relational Concept Analysis (RCA) [1] is an efficient approach to build abstractions in any language, such as modeling languages (UML) or programming languages (Java), that provide specialization-generalization mechanism between different software entities. Abstractions are built for entities using the existing relations between them. Nowadays, there are several tools implementing the RCA process. Starting from contexts and relational contexts, these tools automatically build a set of lattices (called Concept Lattice Family (CLF)) that contains new abstractions.

However, when applied in real case studies, these tools must cope with two main difficult tasks. The first task is the encoding of the model or program to be restructured using a Relational Context Family (RCF), and the second task is the decoding back of the obtained Concept Lattice Family (CLF) into the initial language. For example, several existing methodologies, such as [2,3], propose tools

* France Télécom R&D has supported this work

to encode a UML model into a RCF, then apply a RCA technique, and finally decode back the CLF into a UML model. Note that the use of encoder/decoder is a common problem to FCA and RCA: For each type of model or program one may want to restructure with RCA, a new specific encoder/decoder is needed. To cope with this problem, this paper presents a generic encoder/decoder, whose configuration is based on the structure of the type of model or language to be restructured. This means that, given the elements of the input model or language and the elements that cause the abstractions' building, the repetitive and difficult work in creating RCF and then decoding CLF is fully automatic. For example, in a UML model we will create abstractions for classes and associations, based on the name of the roles of the associations and the name of the classes, whereas in Java we will create abstractions only for classes.

We achieve the design and configuration of the generic encoder/decoder through the use of the Model-Driven Engineering (MDE) paradigm [4], in which every used or produced artifact during the software development is a model. The structure of the model is defined with a metamodel. We proceed in two steps to solve the problem of the generic encoder/decoder. First, we have implemented the RCA process in a MDE-oriented way [2], defining metamodels for RCF and CLF. Second, we have defined a metamodel to define models of the encoding/decoding configuration corresponding to a given language to restructure. To restructure a model, one just has to decide which elements of the model are useful for the RCA-based restructuration, filling up a configuration model. In this paper we focus on the configuration step, i.e. the way we provide a generic encoder/decoder for a RCA-based restructuration.

The rest of the paper is organized as follows. Section 2 first briefly introduces FCA and RCA, and analyses the existing FCA and RCA tools (in particular w.r.t. their capacity to offer generic inputs and outputs). Then, Section 3 gives an overview of Model-Driven Engineering and explains the reasons of the choice of such a paradigm. Then, Section 4 details the mechanisms involved in the generic encoder/decoder for RCA-based model restructuration and illustrates them with UML and Java. Last, Section 5 concludes and gives feedback on the use of MDE for RCA tools.

2 Formal and Relational Concept Analysis

2.1 Background and Definitions

Formal concept analysis (FCA) [5,6,7] is a branch of lattice theory that allows us to identify meaningful groupings of objects that have common attributes.

Definition 1 (Formal Context). *A formal context is a 3-tuple $K = (O, A, I)$, where O and A are finite sets of objects and attributes respectively, and $I \subseteq O \times A$ an incidence relation where $\forall (o, a) \in I$, a is an attribute of the object o.*

With a formal context, several concepts can be produced. A concept is a set of objects that share several attributes. It can be considered as an abstraction of these objects.

Definition 2 (Concept). *A concept is a pair* (X, Y) *with* $X \subseteq O$, $Y \subseteq A$ *and*
$X = \{o \in O | \forall y \in Y, (o, y) \in I\}$ *is the extent (covered objects),*
$Y = \{a \in A | \forall x \in X, (x, a) \in I\}$ *is the intent (shared attributes).*

In other words, the sets of objects and attributes are maximal, i.e. there is no other object 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 objects of the extent. These concepts can be organized in a 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).

FCA is efficient when considering objects described by binary attributes. In order to deal with non binary attributes, FCA has to be extended. One interesting possible extension is to classify several objects taking into account the relations between them. Among the existing approaches, Relational Concept Analysis [1] takes into account this possibility and considers the links between the objects as relational attributes. At the beginning, these relational attributes connect an object to several objects. Then, they connect an object to several concepts that have emerged from the classification of the objects. Instead of having just one formal context, RCA needs to define several formal contexts. Several relational contexts, that represent relations between objects defined in the formal contexts, can be added. Circularity between relations is admitted. Those formal and relational contexts together are called a Relational Context Family (RCF). Figure 1 shows a simple RCF, that deals with animals living in different places. There is a formal context which describes the animals ($K_{animals}$), another one which describes the places (K_{places}), and finally a relational context (R_{live}) describing the living relation between the animals and the places.

Definition 3 (Relational Context Family). *A Relational Concept Family* $\mathcal{R}$ *is a pair* (K, R). *K is a set of formal contexts* $K_i = (O_i, A_i, I_i)$, *R is a set of relational contexts* $R_j = (O_k, O_l, I_j)$ *(*O_k *and* O_l *are object sets of contexts* K_k *and* K_l*).*

Kplaces

	water	earth
River	X	
Sea	X	
Savana		X
Forest		X

Kanimals

	herbivor	carnivor	omnivor
Bear			X
Lion		X	
Crocodile		X	

Rlive

	River	Sea	Savana	Forest
Bear				X
Lion			X	
Crocodile	X			

Fig. 1. A relational context family

New abstractions emerge from a RCF by iterative building of concept lattices, and enhancement of the RCF with the concepts discovered in the concept lattices. The steps of this iterative construction are described below.

Initialization step. The created lattices at this step are the same as the one that would have been created using classical FCA: for each formal context K_i, a lattice $\mathcal{L}_i^0$ is created. Figure 3 (left) shows the initial lattices corresponding to the contexts of the Figure 1.

Step n+1. For each relational context $R_j = (O_k, O_l, I_j)$, a context $R_j^s = (O_k, A, I)$ is created. A contains the extents of the concepts of the lattice $\mathcal{L}_l^n$, and the incidence function I contains the element (o, a) if $S(R(o), a)$ is *true*. The function S is called a scaling operator. The most common scaling operators are $S_\exists(R(o), a)$, that is *true* iff $\exists x \in R(o), x \in a$, and $S_\forall(R(o), a)$, that is *true* iff $\forall x \in R(o), x \in a$. Figure 2 shows such an updated relational context (we used the $S_\exists$ scaling operator). It connects animals with concepts of the lattice (shown in Figure 3 (left)) corresponding to K_{places}. In this lattice, the concept 7 is an abstraction of the places on earth. The application of FCA on $K_k \cup \{R_j^s = (O_k, A, I)\}$ creates new concepts that are added to $\mathcal{L}_k^n$ to obtain $\mathcal{L}_k^{n+1}$. Figure 3 (right) shows such an updated lattice family. $\mathcal{L}_{animals}$ has been calculated using $K_{animals} \cup R_{live}^s$. The concept 10 of $\mathcal{L}_{animals}$ has been created using relational descriptions. It represents the animals that live on earth.

K_{places}

	water	earth
River	X	
Sea	X	
Savana		X
Forest		X

$K_{animals}$

	herbivor	carnivor	omnivor
Bear			X
Lion		X	
Crocodile		X	

R^s_{live}

	Concept_4	Concept_5	Concept_6	Concept_7
Bear	X			X
Lion	X			X
Crocodile	X	X		

Fig. 2. RCF of Fig. 1 after the first step

This process stops when all the lattices of a step n are isomorphic to those of the step $n-1$. After this step, the relational contexts will not be modified and no more concepts will emerge.

2.2 Encoding and decoding input data in FCA/RCA-based tools

In order to understand the existing limitations of FCA/RCA-based tools, an overview of the existing ones is needed taking into account their encoding/decoding operation. This operation, named *FCA mapping*, consists in converting input data into FCA contexts and then converting the built lattices back to the initial format. Most of the existing FCA-based tools [8,9] do not consider the FCA mapping as a crucial task. However, if a tool builder wants to integrate any FCA-based tool, he has to build himself a mapping tool to convert the input data to be analyzed into the input format of the FCA-based tool. Such an approach is expensive, moreover the knowledge of the configuration is included in the mapping tool, and the code of this tool has to be modified if the configuration changes.

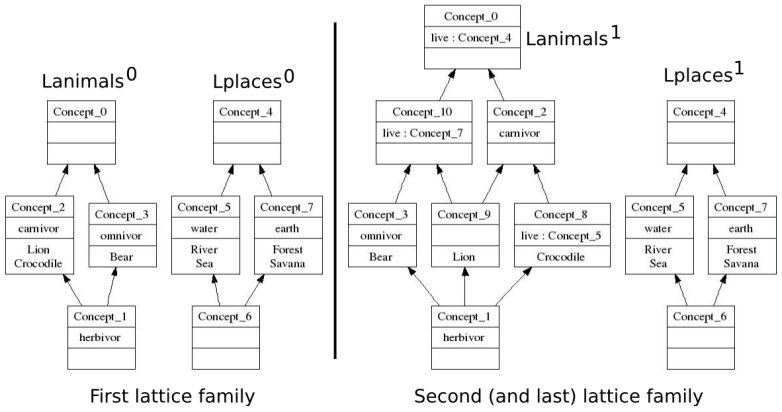

Fig. 3. The lattice families corresponding to the RCF of Fig. 1 and Fig. 2

Some FCA-based tools like *ConAn* [10](a FCA-based reengineering tool) have identified the importance of defining FCA mappings, and the advantages and drawbacks respectively. *ConAn* uses a software model expressed in a language independent way (called a FAMIX model), and FCA mappings to build a formal context. After building the corresponding lattices, *ConAn* builds a new model, called a high-level view (to help in the software analysis) in a format understandable by a regular developer with no lattice theory knowledge. Unfortunately, *ConAn* is focused on software reengineering and does not provide facilities to express neither FCA mappings nor high-level view formats. It is still necessary to hand-code a translator between a base model (named as FAMIX) and a formal context and then, between the concept lattices and the high-level view.

From our viewpoint, the most advanced approaches are [3] and [11]. They aim at discovering new abstractions in UML models using RCA. They use different tools: (1) a CASE tool (Objecteering [12] for [3], Eclipse [13] for [11]) to generate a context family, (2) *Galicia* [14] to build the corresponding lattice family, and then, (3) the CASE tool again to build an improved UML model. The FCA mapping between a UML model and a context family can be configured in both tools. However, this configuration mechanism is specific to UML. Therefore these tools cannot be used to analyze other input data, such as Java source code.

A FCA-based tool to easily define FCA mappings on a wide range of input data formats (UML model, Java code, Smalltalk code, ...) is still missing. The design of such a tool is the contribution of this paper. To realize it, we need to introduce a high-level configuration mechanism which allows to express which elements of the input data have to be transformed into FCA items. For that purpose, it is necessary to use a high-level modeling language, which is able to represent a UML model or a Java source code in a common way. Such support is provided by the Model Driven Engineering paradigm, described in the next section.

3 Model Driven Engineering

Model Driven Engineering [4] 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 metamodel (a model is said to conform to a metamodel). To pragmatically handle two models that conform to two different metamodels (for example to transform a UML model into a Relational Database model), a program has to be written, dealing with both metamodels. For that purpose, MDE assumes the existence of a unique meta-metamodel. Such a meta-metamodel allows to define how a metamodel is structured. Mainly, two meta-metamodels are used: EMOF [15] (defined by the OMG) and Ecore [16], (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 4), and is expressive enough to describe itself. Therefore a meta-meta-metamodel is not necessary.

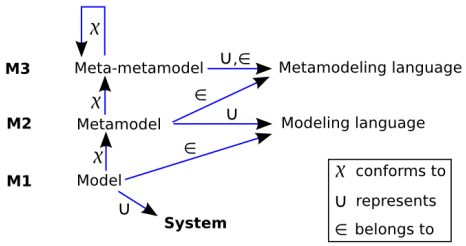

Fig. 4. The metamodeling hierarchy

A fundamental element of MDE is the notion of *model transformation*. A model transformation is a program or a set of rules that takes one or several models in input (conform to one or several metamodels) and produces as output one or several models.

In Section 1, we explained that the goal of this paper is to provide a high-level configuration mechanism allowing to define FCA mappings. Such a mechanism must be able to define FCA mappings, for example, from a UML model as well as from Java code, because Ecore can describe either a UML model or Java code. Therefore, a FCA mapping can be considered as a model transformation based on Ecore (and thus handling all kinds of models) from a model into a RCF model, and reciprocally from a CLF model to an output model (we will introduce what is a RCF and CLF model in the next section). Since all the information from a UML model or Java code is not relevant for a RCA restructuration, the tool builder still have to be able to select what kind of data he wants to place in the FCA items, that will be produced by the transformation. The next section

explains our approach in detail, based on MDE, that allows an easy definition of FCA mappings.

4 FCA/RCA with a MDE approach

A global view of a FCA/RCA process using a MDE approach is shown in Figure 5. The purpose of using such a process on an input data (called model in the MDE approach) is to lead to the creation of output data that contains relevant abstractions. In this process, we consider that input and output models conform to the same metamodel. This is the most common case when using a RCA process for restructuration or abstraction discovery.

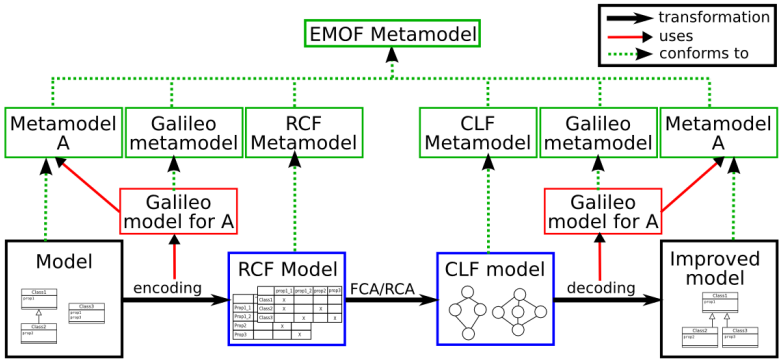

Fig. 5. Process overview

The *encoding* transformation of the process aims at producing FCA/RCA items from an input model. Only some configuration data is required in order to perform that transformation. The FCA/RCA transformation (described in section 2) builds the lattices corresponding to the input contexts. The *decoding* transformation consists in converting the produced lattices back to output data, conforming to the same metamodel as that before the *encoding* transformation. Since the input and output data metamodels are the same, the *decoding* transformation uses the same configuration as the *encoding* one to produce the output model.

To illustrate how the *decoding* and the *encoding* steps work, we use two sample input models: a UML model and a Java source code. Figure 6 shows these models in a MDE fashion, detailing models, metamodels and the meta-metamodel. The *conforms to* arrows show the connections between the elements of the models and the elements of the metamodel; and between the metamodels and the meta-metamodel. For the sake of clarity, the Java, UML and Ecore metamodels have been presented in a reduced form. Next we describe how these models are transformed through the *encoding* and *decoding* operations.

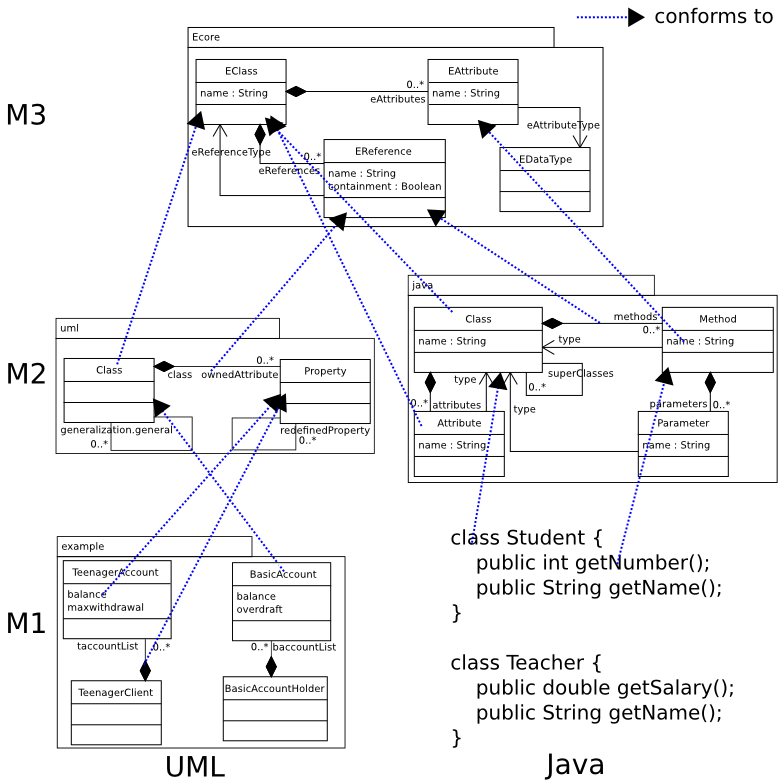

Fig. 6. The sample models

To remain in a MDE approach, a RCF metamodel was created, which is is able to define data structures similar to the one described in Section 2. Figure 7 shows this metamodel: An RCF is composed of several formal contexts (called *EntityAttributeContext* in the metamodel) and several relational contexts (called *InterEntityContext*). These contexts contain an incidence relation (represented as a set of *Pair* in the metamodel).

We use an UML and a Java model. In this restricted example, we are interested in using RCA on the UML model to create new superclasses by factorizing the properties of the existing classes, based on their names. For Java, we want to use RCA in order to create new superclasses that factorize methods from the existing classes, based on their names. To achieve these goals, the *encoding* operation has to generate formal and relational contexts describing the previously quoted elements. We introduce a configuration metamodel to give this information to the transformation.

Figure 8 shows this metamodel. Models conform to this configuration metamodel are used in the *encoding* transformation to dynamically choose the elements to consider and how to encode them. Figure 9 shows configuration models

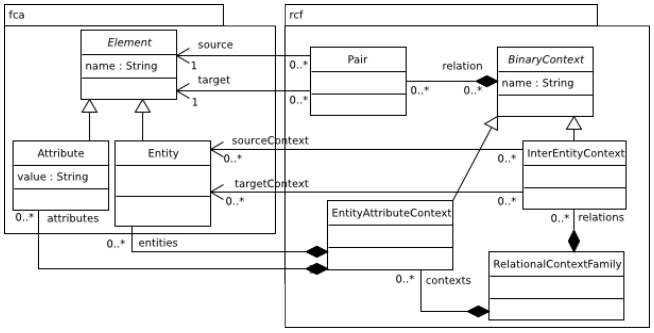

Fig. 7. The Relational Context Family metamodel

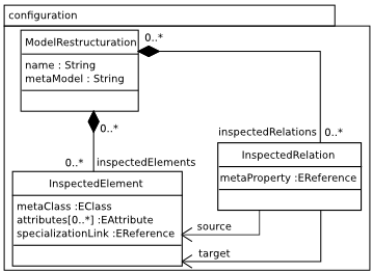

Fig. 8. The encoding/decoding configuration metamodel

for UML and Java that lead to the creation of contexts describing the elements we want to analyze.

The *encoding* transformation works as follows. First, several sets of elements lead to the creation of several formal contexts, one for each kind of element. The *InspectedElement* elements coming from the configuration model parameterize this stage of the transformation. For each *InspectedElement*, a set of elements is dynamically selected via the *metaClass* information. The binary attributes that describe this set of elements are selected via the *attributes* information. A formal context is finally created using these elements as objects and the binary attributes. Given the configurations plotted by Figure 9, two formal contexts are created for the UML model: one describing the classes and the other one, the properties (using the *name* as attribute). For the Java code, two formal contexts are created: one for the classes, the other one for the methods (using *name* as attribute). Figure 10 and 11 show the formal contexts generated from the examples.

Secondly, several relations between the elements of the input model are transformed into several relational contexts. The *InspectedRelation* elements coming from the configuration model parameterize this stage of the transformation. For each *InspectedRelation*, a relation is selected in the input model via the *metaProperty* information. The *source* and *target* are used to select the elements

```
Config. for UML Class Model restructuring:

Inspected Elements:
Class: attributes = [],
 specializationLink = "generalization.general"
Property: attributes = ["name"],
 specializationLink = "redefinedProperty"

Inspected Relations:
ownedAttribute: source = Class,
 target = Property
type: source = Property,
 target = Class
```

```
Config. for Java source code restructuration:

Inspected Elements:
Class: attributes = [],
 specializationLink = "generalization.general"
Method: attribute = ["name"]

Inspected Relations:
methods: source = Class, target = Method
```

Fig. 9. Two configuration models

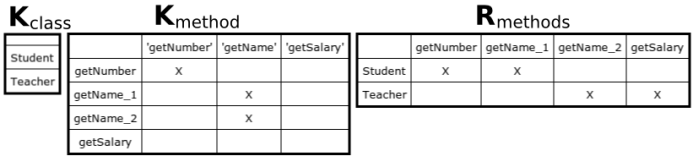

Fig. 10. The generated Java contexts

(described in the previously built formal contexts) involved in this relational context. With the configuration in Figure 9, two relational contexts are created for the UML model: one for the *ownedAttribute* relation between the classes and the properties, and the other for the *type* relation between the properties and the classes. For the Java code, one relational context is created to describe the *methods* relation between the classes and the methods. Figures 10 and 11 show relational contexts generated from the examples.

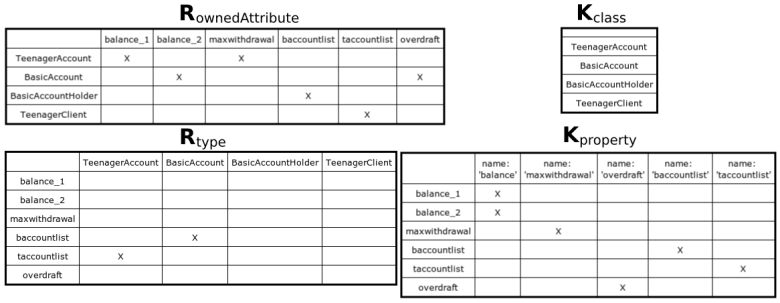

Fig. 11. The generated UML contexts

The *decoding* transformation aims at producing a model using the concept lattices produced with the RCA. Figure 12 shows the metamodel of a concept lattice family. This transformation uses the same configuration model as the *decoding* transformation. The input is a family of concept lattices, one for each

context of the input RCF. These contexts are used to represent a kind of objects present in the source model. For the UML model, there is one lattice for the classes and one for the properties. For the Java code, there is one lattice for the classes and one for the methods. Therefore, the elements built from a lattice have the same metaclass as the elements that lead to the creation of this lattice. For instance with UML, the class lattice coming from the class formal context produces classes in the ouput UML model. The partial order between the concepts of a lattice is used to create relations between the elements generated from the concepts of this lattice. The name of the relation to create is in the *specializationLink* information from the *InspectedElement* coming from the configuration model associated with the lattice. With UML, the partial order between the concepts of the class lattice is interpreted as a *generalization.general* between the classes. Binary attributes and relational attributes from a concept of a lattice need to be placed in the corresponding elements. The binary attributes lead to the creation of attributes in the corresponding element. Relational attributes generate relations in the output model between the element corresponding to the concept and the element corresponding to the target concept of the relational attribute.

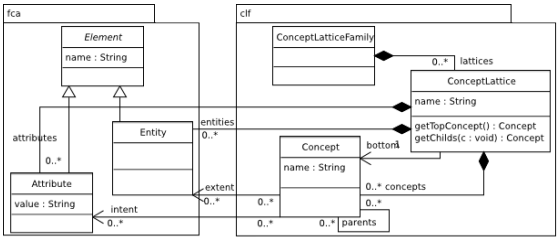

Fig. 12. The Concept Lattice Family metamodel

5 Conclusions and Future Work

We have proposed a generic way to encode models and programs so that they can easily benefit from a restructuration based on FCA or RCA, as well as the reciprocal generic way to decode the obtained lattices. The genericity of the approach has been designed using the Model-Driven Engineering paradigm, and is based on the knowledge of the metamodels of the artifacts to restructure and the underlying common meta-metamodel. To restructure a model, one has just to give (in addition to the model itself): the metamodel of the model, and a configuration model making precise which kinds of elements of the input model are to be taken into account. This approach is implemented in a tool written in Java EMF that has been experimented with UML models (from France Télécom projects [17]), Java programs, OWL, and Ecore models.

MDE gave us further significant benefits. The parameters of the RCA process have been identified and properly modeled, consequently the RCA restructuration can be easily fine-tuned. We plan to go a step further in the parameterization

of the RCA process, allowing the user to choose the algorithm to build lattices and the scaling operator, or to define the number of steps he wants the algorithm to perform.

Further work will consist in adapting our approach so that the input metamodel used for the encoding can be different from the output metamodel used for the decoding. This is useful when the discovered abstractions cannot be expressed with the input metamodel (but could be expressed with a similar metamodel). For example, restructuring Java programs may create multiple class inheritance, that cannot be represented with a Java program, but that can be represented with a UML class model.

References

1. Huchard, M., Hacene, M.R., Roume, C., Valtchev, P.: Relational concept discovery in structured datasets. Ann. Math. Artif. Intell. **49**(1-4) (2007) 39–76
2. Arévalo, G., Falleri, J.R., Huchard, M., Nebut, C.: Building Abstractions in Class Models: Formal Concept Analysis in a Model-Driven Approach. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: MoDELS. Volume 4199 of Lecture Notes in Computer Science., Springer (2006) 513–527
3. Dao, M., Huchard, M., Hacene, M.R., Roume, C., Valtchev, P.: Towards Practical Tools for Mining Abstractions in UML Models. In Manolopoulos, Y., Filipe, J., Constantopoulos, P., Cordeiro, J., eds.: ICEIS (3). (2006) 276–283
4. Kent, S.: Model Driven Engineering. In Butler, M.J., Petre, L., Sere, K., eds.: IFM. Volume 2335 of Lecture Notes in Computer Science., Springer (2002) 286–298
5. Birkhoff, G.: Lattice theory. New York (1948) 247
6. Barbut, M., Monjardet, B.: Ordre et Classification: Algèbre et Combinatoire. Volume 2. Hachette (1970)
7. Ganter, B., Wille, R.: Formal Concept Analysis, Mathematical Foundations. Springer, Berlin (1999)
8. Conexp: Conexp. http://conexp.sourceforge.net/ (2006)
9. Burmeister, P.: Formal Concept Analysis with ConImp: Introduction to the Basic Features. Fachbereich Mathematik, Technische Universität Darmstadt (2003)
10. Arévalo, G., Ducasse, S., Nierstrasz, O.: Lessons Learned in Applying Formal Concept Analysis to Reverse Engineering. In Ganter, B., Godin, R., eds.: ICFCA. Volume 3403 of Lecture Notes in Computer Science., Springer (2005) 95–112
11. Seuring, P.: Design and implementation of a UML model refactoring tool. Master's thesis, Hasso-Plattner-Institute for Software Systems Engineering at the University of Postdam (2005) http://www.lirmm.fr/~huchard/Documents/Papiers/PhilippSeuringMasterThesis.pdf.
12. Softeam: Objecteering. http://www.objecteering.com/ (2007)
13. Eclipse: Eclipse. http://www.eclipse.org/ (2007)
14. Galicia: Galicia. http://www.iro.umontreal.ca/ galicia/ (2005)
15. OMG: MOF 2.0 core specification. http://www.omg.org/cgi-bin/doc?ptc/2004-10-15 (2004)
16. Eclipse: The Eclipse Modeling Framework (EMF) Overview. http://www.eclipse.org/emf/docs.php?doc=references/overview/EMF.html (2005)
17. Falleri, J.R., Arévalo, G., Huchard, M., Nebut, C.: Etude de l'extension d'une méthode de restructuration de diagrammes UML/ Tests sur la plate-forme Petra. Contrat particulier de Recherche Externalisée 5326. Tâche 2. LIRMM/France Télécom (2007)