Building Abstractions in Class Models: Formal Concept Analysis in a Model-Driven Approach

Gabriela Beatriz Arévalo, Jean-Rémy Falleri, Marianne Huchard, Clémentine Nebut

Building Abstractions in Class Models: Formal Concept Analysis in a Model-Driven Approach

Gabriela Arévalo, Jean-Rémi Falleri, Marianne Huchard and Clémentine Nebut

LIRMM, CNRS and Université de Montpellier 2,
161, rue Ada, 34392 Montpellier cedex 5, France
{arevalo, falleri, huchard, nebut}@lirmm.fr

Abstract. Designing class models is usually an iterative process to detect how to express, for a specific domain, the adequate concepts and their relationships. During those iterations, the abstraction of concepts and relationships is an important step. In this paper, we propose to automate this abstraction process using techniques based on Formal Concept Analysis in a model-driven context. Using UML2.0 class diagrams as modeling language for class models, in this proposal we show how our model-driven approach enables parameterization, tracing and generalization to any metamodel to express class models.

1 Introduction

In model-driven development, modeling activities have as purpose (at least partially) to replace the coding tasks. Unfortunately, the model engineer does not have all the same facilities (such as versioning and refactoring tools) as in mostly classical coding environments. With these kinds of tools, the model-driven paradigm could be adopted in large software companies. Specifically, within the context of refactoring object-oriented models, in this paper we focus on automating the detection and building of class hierarchies. Designing class models is not a trivial task. It is an iterative process to detect how to express, for a specific domain, the adequate concepts and their relationships. During this iterative process, the abstraction of concepts and relationships is a crucial task. Indeed, abstraction provides better concept structuring and more reusable artifacts. In this paper, we propose to automate this abstraction process using an adaptation of Formal Concept Analysis (FCA) techniques [1] in a model-driven context. FCA has proved to be an efficient technique to build or restructure class hierarchies [2,3,4], but has not been yet applied in a model-driven approach.

The contribution of this paper is a FCA-based model-driven approach to abstract concepts involved in a class model (classes, associations, attributes and so on). Briefly, this process uses the successive application of model transformations as a main building mechanism. We use two main tools: Kermeta [5] and UML. Using Kermeta [5] (compatible with MOF and OCL) as our metamodeling language, we are able to (1) give an operational semantics to every underlying metamodel and implement every model transformation, and (2) describe the FCA algorithms and check their performances. Using the UML as a language, we describe class models. As a result, the transformations are defined based on a part of the UML 2.0 metamodel. However, the specification and implementation of our proposal using model transformations turns to be easily tunable by parameters, and applicable to other metamodels which handle adequate concepts to detect and build abstractions. Our approach shows that formalizing FCA with model transformations gives interesting benefits, such as tracing the different steps of the process, or the parameterization. These characteristics are also important if we compare our contribution to the one introduced in [6]. In that approach the main limitation was that the authors consider the model transformations as a black box, with no means of tracing or parameterizing.

The paper is structured as follows. Section 2 gives a brief overview of our approach, recalls the main notions of FCA, and introduces the example used all over the paper. Each main transformation is then detailed into Sections 3, 4 and 5 respectively. Section 6 discusses the benefits and limitations of this approach, as well as related work.

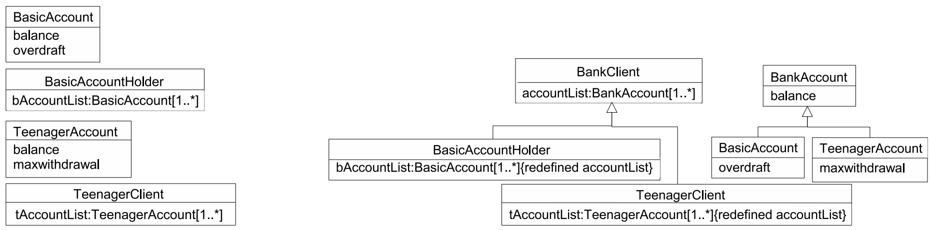
2 Overview and background

Building class models is usually not a trivial task but rather an iterative process aiming at finding the simplest model with good properties such as, for example, maintainability, adequate factorization and easy testing. While building a class model, one task consists in generalizing concepts: finding regularities in already identified concepts in order to detect new abstractions. When representing class models with UML class diagrams, several model elements can be abstracted such as, obviously, classes, but also associations, attributes, and methods. As an example, starting from the class model shown in Fig. 1(a), the class model of Fig. 1(b) can be obtained, where new classes have been introduced (for example class `BankClient` that is an abstraction of the `BasicAccountHolder` and the `TeenagerClient` classes), as well as new attributes (e.g. the attribute `accountList` that abstracts the two attributes `bAccountList` and `tAccountList`). Our approach aims at automating this refactoring, i.e. at detecting and building new abstractions in a class model, using Formal Concept Analysis (FCA). Before going into the details, we provide in this section the minimal notions of FCA, and then we give an overview of our approach, that will be detailed in the next sections.

2.1 Background on FCA

FCA [1] is a mathematical technique, based on lattice theory, to discover abstractions (known as *concepts*) from a set of entities (formal objects) described by attributes (formal attributes) [1]. Concept specialization draws a lattice structure. Basic FCA considers *formal contexts* $\mathcal{K} = (E, P, I)$ as shown in Figure 2

[1] All over the text we use the term attributes to denote formal attributes, except in case we must clarify the ambiguity between attributes of a class model and formal attributes of a FCA context.



(a) A simple class diagram

(b) Refactored class diagram

Fig. 1. The example of bank accounts

(left). E is the entity set (here UML classes), P the attribute set (here UML attributes) and I associates an entity with its attributes: $(e, p) \in I$ when entity e owns attribute p. With any entity set $X \subseteq E$ we associate the shared attributes with the mapping α defined by $\alpha(X) = \{p \in P \mid \forall e \in X,\ (e, p) \in I\}$. Symmetrically, with any attribute set $Y \subseteq P$ we associate the entities owning all the attributes of Y. To that end, we use the mapping ω defined by $\omega(Y) = \{e \in E \mid \forall p \in Y,\ (e, p) \in I\}$. In the example, let $Y = \{balance\}$, we have $\omega(Y) = \{BasicAccount, TeenagerAccount\}$, while for $X = \{BasicAccount\}$, $\alpha(X) = \{balance, overdraft\}$. A concept is a pair (X, Y) where $X \subseteq E$, $Y \subseteq P$, $\alpha(X) = Y$ and $\omega(Y) = X$. In Figure 2, $\{\{BasicAccount, TeenagerAccount\}, \{balance\}\}$ is a concept. Graphically, this concept corresponds to the vertical block in the column *balance*. More generally, a concept corresponds to a block of maximal size in the context (the blocks are found in the context modulo the order of the columns and rows). X (resp. Y) is usually called the extent (resp. intent) of the concept.

The specialization order between concepts corresponds to extent inclusion (or intent containment). The concept lattice $\mathcal{L} = (\mathcal{C}, \leq_{\mathcal{L}})$ is the set of concepts provided with the inclusion partial order. In Figure 2, the concept $\{\{BasicAccount\}$, $\{balance, overdraft\}\}$ specializes the concept $\{\{BasicAccount, TeenagerAc-$ $count\}$, $\{balance\}\}$.

The concept at the bottom has no interest as it represents the hypothetic set of entities containing all attributes. The concepts at the first level correspond to initial classes. The unique concept of the second level stems from the factorization of property `balance`. In our example, it could generate a new UML class factorizing `balance` and appearing as a superclass of `BasicAccount` and `TeenagerAccount` (class `BankAccount`). The top concept gathers attributes common to all entities, in this specific case it is an empty set of attributes. This lattice is very simple, but in general, systematic factorization in real software projects generates too many concepts, which makes the analysis difficult to grasp. The main advantage of using FCA for UML class diagram reconstruction is that we

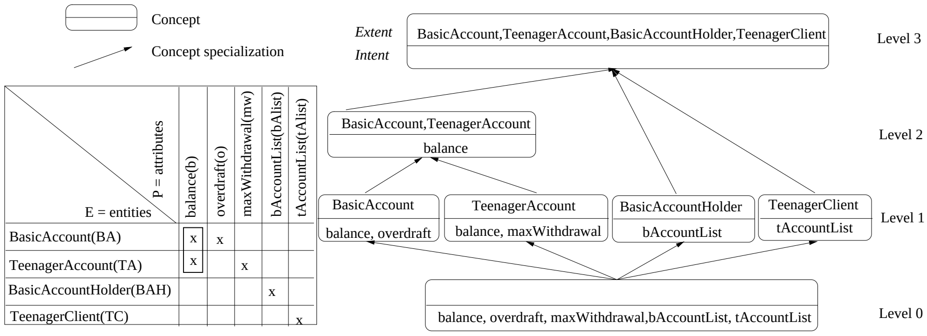


Fig. 2. A context $\mathcal{K}$(left) and the lattice (right) describing bank accounts.

obtain a sort of normal form for class models. In this normal form, redundancy is eliminated (total factorization is achieved) and the specialization order between classes exactly matches the inclusion order between property set of the classes. Besides that, maximal factorization is obtained with minimal number of classes.

However, even in this very simple example, relevant abstractions remain undiscovered by this naive process. Let's see carefully at the two attributes `bAccountList` and `tAccountList`. Their types, respectively `BasicAccount` and `TeenagerAccount`, are evidently generalizable by a class such as `BankAccount` factorizing `balance`. Thus, the idea is to continue the process and decide that `bAccountList` and `tAccountList` share a common abstraction, namely *list of accounts*. To discover that abstraction, we need to go further into the representation of the UML class diagram, giving the status of entities to UML properties. As a result, UML classes and UML properties are described by characteristics including property ownership and classes used as types for properties. In the following section we explain how an extension to the theory of Formal Concept Analysis, named *Relational Concept Analysis* (RCA), allows such information to be treated.

2.2 Class hierarchy refactoring using FCA in a MDE context

Figure 3 shows an overview of our approach consisting of 3 model transformations[2].

1. The first transformation, *UML2Contexts*, turns the original UML 2.0 class diagram into a set of binary contexts and binary relations. It is a transformation from a UML 2.0 metamodel [7] to a relational context family metamodel.

[2] All over this paper, we use an object terminology to refer to model conformance, for example we talk about models that are instances of meta-models. It can be seen as a terminological misuse, but since we are working with an object-oriented language (Kermeta [5]) to define the metamodels and the model transformations, this terminology is the most adequate one to our work.

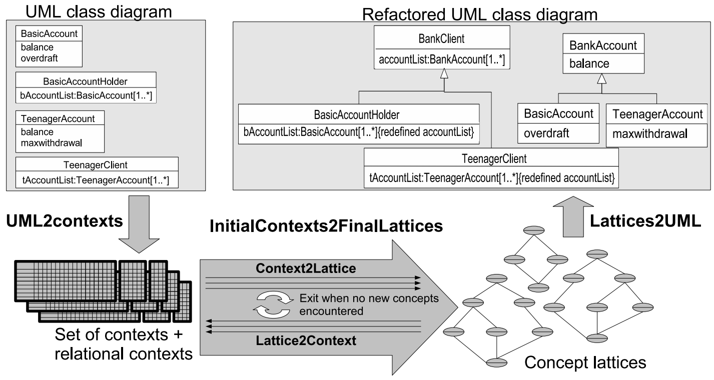


Fig. 3. Overview of our approach

2. The second transformation, *InitialContexts2FinalLattices*, aims at obtaining a set of concept lattices of the final class diagram from the initial set of contexts. It is a transformation from a relational context family metamodel to a concept lattice family metamodel.
3. The third transformation, *Lattices2UML* consists in translating the obtained concept lattices into a UML 2.0 class diagram using traceability information from the previous transformations.

Using a model-driven approach based on Formal Concept Analysis in order to refactor models is very fruitful. First, it allows to define a simple sequence of model transformations (in particular for the second transformation) without using a complex algorithm. Second, the proposed approach can be applied to classify any kind of concepts as soon as they are defined by a metamodel. Indeed, the core of the approach is the second transformation, and adapting the approach to another metamodel only requires to develop new transformations to replace the first (*UML2Contexts*) and the third (*Lattices2UML*) ones. As we have said in Section 1, every step of the approach is automated and every transformation is implemented in Kermeta [5].

3 From UML to formal contexts

In this section, we detail the transformation from a UML model to formal contexts handled by Relational Concept Analysis.

3.1 Metamodels involved in the transformation

In our approach we use the small metamodel deduced from the UML 2.0 metamodel (shown in Figure 4) to express class models. Working with such a reduced metamodel is not restrictive, since applying work on model typing and model type substitutability presented in [8], we can use a model conform to the whole UML 2.0 metamodel as an entry model of our transformation. In the rest of the paper, we will refer indifferently to the UML 2.0 metamodel or its reduced form. We focus only on classes, attributes and associations in the framework of our example: attribute `name`, class `Class`, class `Property`, role `type` which associates their type to properties and role `ownedAttribute` which associates their attributes to classes. As a simplification, we have restricted the end of role `type` to be `Class` rather than `Type`, a superclass of `Class`. `ownedAttribute` is in fact a derived role in the original UML 2.0 metamodel and we consider only flattened models (without inheritance relationships, just for simplification reasons). `ClassHierarchy` is used as an entry point in the models, while the derived role `superclass` and the role `redefinedProperty` are used only in the third transformation.

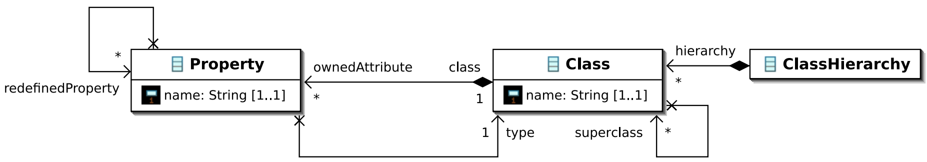


Fig. 4. Adaptation of a restriction of the UML metamodel

Relational Concept Analysis [6] considers a family of contexts rather than a single one, allowing to separate entities into several categories. In our example, there are two categories: `Class` and `Property` (see the example of RCF in Figure 6). The contexts of a family include relations that link entities of one kind to entities of another kind. Those relations come from the associations in the underlying metamodel (here the UML 2.0 metamodel, see Fig. 4). In our example, we deal with two relations: `ownedAttribute` and `type`. This set of contexts together with the relations is called a Relational Context Family (RCF). The associated metamodel is given in Figure 5. More formally, a relational context family $\mathcal{F}$ is a pair $(\mathcal{K}, \mathcal{R})$ where:

- $\mathcal{K}$ is a set of *contexts* $K_t = (E_t, P_t, I_t)$ linking entities to attributes (`EntityAttributeContext` in Fig. 5). In our example $\mathcal{K} = \{K_{Class}, K_{Property}\}$.
- $\mathcal{R}$ is a set of contexts R_s expressing *relations* between entities coming from different contexts of $\mathcal{K}$. R_s is such that $\exists\ K_{t1},\ K_{t2} \in \mathcal{K},\ R_s \subseteq E_{t1} \times E_{t2}$. R_s is represented by `InterEntityContext` in Fig. 5. In the following, those contexts will be denoted as relations. In our example, $\mathcal{R} = \{R_{ownedAttribute}, R_{type}\}$ where $R_{ownedAttribute} \subseteq E_{Class} \times E_{Property}$ and $R_{type} \subseteq E_{Property} \times E_{Class}$.

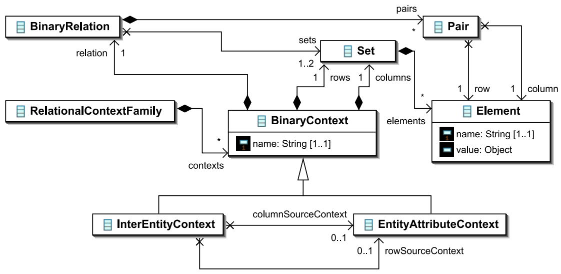


Fig. 5. The Relational Context Family (RCF) metamodel

3.2 The transformation from UML to a family of contexts

We here explain how a UML model is automatically transformed into a relational context family. To illustrate this transformation, the result of its application on the UML class diagram of Figure 1(a) is shown in Figure 6. The Relational Con-

K_{Class}	name= "BasicAccount"	name= "TeenagerAccount"	name= "BasicAccountHolder"	name= "TeenagerClient"
BA	X			
TA		X		
BAH			X	
TC				X

$K_{Property}$	name= "balance"	name= "overdraft"	name= "maxWithdrawal"	name= "bAccountList"	name= "tAccountList"
bba	X				
bta	X				
o		X			
mw			X		
bAlist				X	
tAList					X

R_{type}	BA	TA	BAH	TC
bba				
bta				
o				
mw				
bAlist	X			
tAlist		X		

$R_{ownedAttribute}$	bba	bta	o	mw	bAlist	tAlist
BA	X		X			
TA		X		X		
BAH					X	
TC						X

Fig. 6. The Relational Context Family obtained from the UML model of Figure 1(a)

text Family is automatically deduced from the UML 2.0 metamodel as follows (we discuss only our restricted case but the principle is the same on the whole UML 2.0 metamodel).

- Selected metaclasses of the source metamodel (here: UML) give rise to contexts: in our example, $\mathcal{K}$ is composed of the two contexts, K_{Class} and $K_{Property}$ (as shown in Figure 6). Pairs composed of selected meta-attributes of these classes and their values on the studied model are transformed into the formal attributes in the target contexts. In our example, pairs are formed with the meta-attribute `name`.

– Relations of $\mathcal{R}$ come from selected roles in the associations of the source metamodel. In our example, we obtain the two relations R_{type} and $R_{ownedAttribute}$ shown in Figure 6. Values for all the relations are deduced from a view of the studied model as an instantiation of the UML metamodel (see the object diagram of Figure 8).

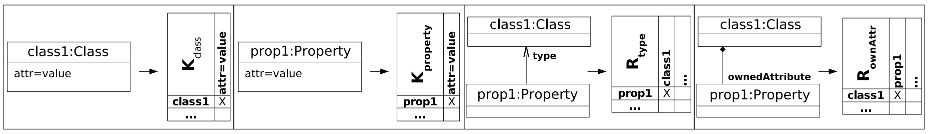


Fig. 7. Transformation from UML to context

Those two transformation rules are illustrated in Figure 7.

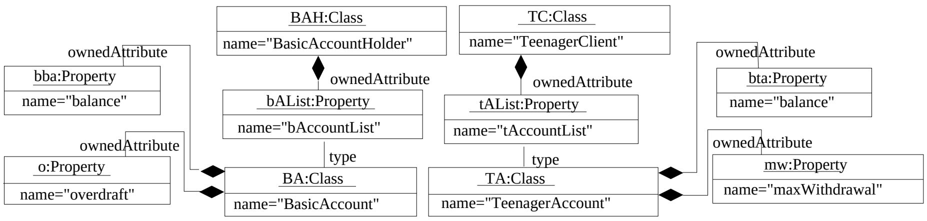


Fig. 8. Our class model of Fig. 1(a) as an instantiation of the simplified UML metamodel

Part of the relevance of this transformation relies on the possibility to fine-tune it. Choosing UML metamodel classes, attributes and associations to be encoded in the RCF is a delicate task. Some model elements provide quite technical information, such as multiplicity or visibility, while others expose the semantics of the domain such as names in general. For example, we do not want to generalize two classes or two associations because they are both abstract. As a result, we do not take into account the meta-attribute `isAbstract` of the UML metaclass `Classifier` during the generalization process.

4 Class hierarchy refactoring: Iterative Transformation

In this section, we describe the core transformation of our approach, named *InitialContext2FinalLattices*, that aims at generating the lattice models from the initial Relational Context Family (RCF). The metamodel for the lattices is given in Figure 9.

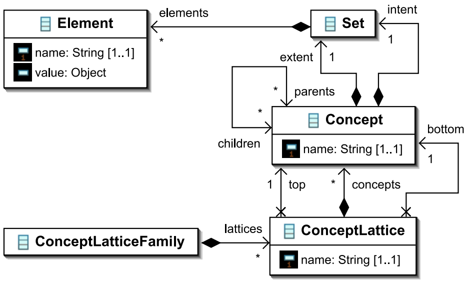


Fig. 9. The metamodel for lattices

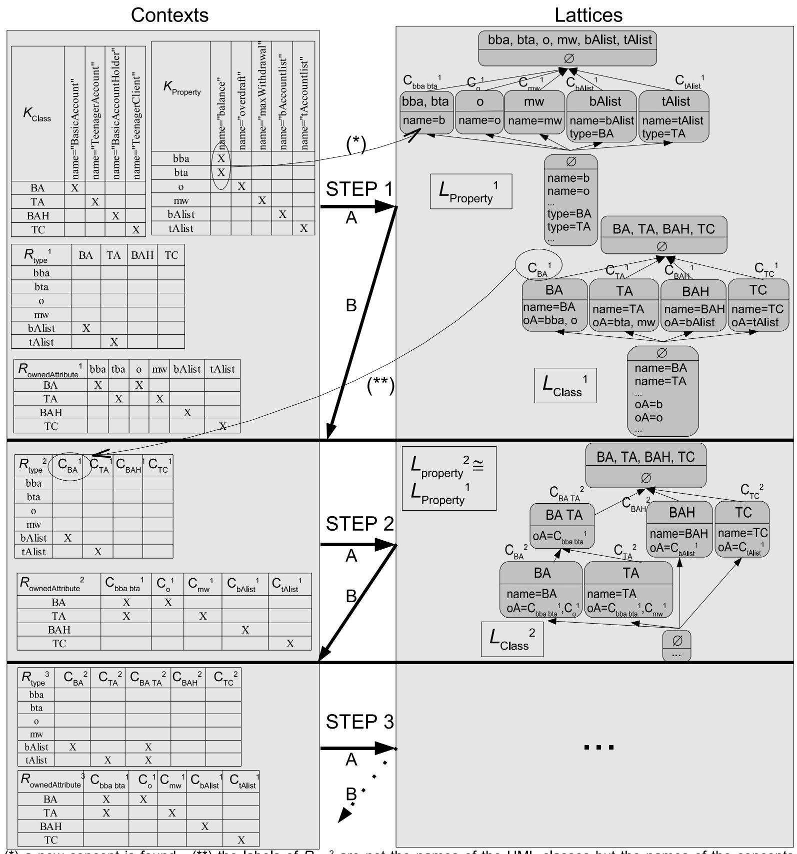
A family of lattices is composed of concept lattices. The concepts of a lattice are ordered by the specialization relation represented by the association `children`/`parents`. A concept is composed of an `extent` and an `intent` that are two sets of elements.

This transformation (summarized in the bottom of Figure 3, and applied on our example in Figure 10) consists in iterating on the multiple application of two smaller transformations, *context2lattice* and *lattice2context*. Indeed, processing a RCF involves alternative construction of lattices (one per context) and enrichment of the relations $\mathcal{R}$ of the RCF by knowledge coming from lattices. The process stops when a fix point on lattice construction is reached, namely when no new abstraction emerges.

More precisely, we define a step of the transformation *InitialContext2FinalLattices* as a multiple application (one application per context) of the transformation *context2lattice* (part A of the step) followed by a multiple application (one application per target relation) of the transformation *lattice2context* (part B of the step). In the bottom of Fig. 3 and in Figure 10, a step corresponds to a round-trip (A followed by B). The initial RCF is named RCF^1 and owns contexts and relations also numbered 1. RCF^1 generates in step 1 (A) lattices numbered 1 with concepts numbered 1, then those lattices generate in step 1 (B) a new RCF numbered RCF^2 and so on. This iteration stops when no concept is found during a step.

Part A of step i. The multiple application of the sub-transformation *context2lattice* builds one lattice for each entity-attribute context of RCF^i. The source model of *context2lattice* is a context extended by all the relations with the same entity set. More formally, the source model is a context $K_p = (E_p, P_p, I_p)$ extended by all relations $R^i \in RCF^i$ such that $R^i \subseteq E_p \times Y$ (Y is either an entity set E_q at step 1, or the concept set of a lattice at step i, $i > 1$). The rule of this transformation is illustrated in Figure 11. For example, the K_{class} context is extended by the relation $R^i_{OwnedAttribute}$, while the $K_{Property}$ context is extended by the relation R^i_{type}. The transformation consists in building a lattice following classical Formal Concept Analysis. At this step i, the target model (i.e. the lattice model) obtained from the extended context K_p is denoted $\mathcal{L}^i_p = (X^i_p, \leq_{\mathcal{L}^i_p})$ where X^i_p is the set of concepts and $\leq_{\mathcal{L}^i_p}$ is the specialization order.

Part B of step i. The multiple application of the sub-transformations *lattice2context* builds a set of relations (initial contexts – in our example K_{Class} and $K_{Property}$ – are not modified during this transformation). During a *lat-*



(*) a new concept is found - (**) the labels of R_{type}^2 are not the names of the UML classes but the names of the concepts, found during step 1, that generalize the classes - The relation ≅ between 2 lattices means that the lattices are identical except that in their extents, the concepts are differently numbered.

Fig. 10. Iterative transformation applied to the accounts example

tice2context execution, a relation $R^{i+1} \subseteq E_p \times X_q^i$ is generated. The principle is to replace labels of columns in initial relations by concepts. The rules of this transformation are shown in Figure 12. Let us consider the relation $R_j^1 \subseteq E_p \times E_q$. During part B of step i, R_j^1 is replaced by $R_j^{i+1} \subseteq E_p \times X_q^i$, with $(e, C_f) \in R_j^{i+1}$ if $(e, f) \in R_j^1$ and $f \in Extent(C_f)$. For example, during part B of step 1, the labels of the columns of $R^1_{ownedAttribute}$ are replaced by the concepts of the lattice $\mathcal{L}^1_{Property}$ (see Figure 10). We have $(BA, C^1_{bbabta}) \in R^2_{ownedAttribute}$ since $(BA, bba) \in R^1_{ownedAttribute}$ and $bba \in Extent(C^1_{bbabta})$. An interpretation is that C^1_{bbabta} is a generalization of *bba*, more precisely an abstraction of properties named "balance". Moreover, class BA owns *bba*, then BA owns *bba* generalizations, including C^1_{bbabta}. At the end of this transformation, each lattice is associated with a context (via traceability links) and by construction to a

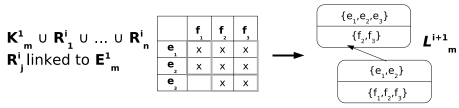


Fig. 11. Transformation rule for *context2lattice*

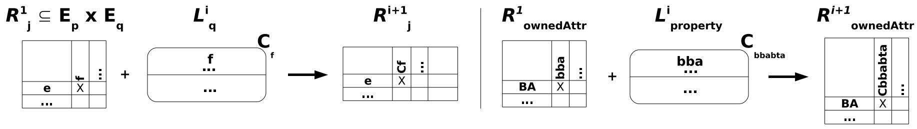


Fig. 12. Transformation rules for *lattice2context*

class of the UML metamodel; in our example, lattices $\mathcal{L}_{Class}$ and $\mathcal{L}_{Property}$ are associated with metaclasses `Class` and `Property`.

5 Effective refactoring : coming back to the UML

Our last transformation, *FinalLattices2UML*, parses lattices and generates UML elements. This transformation was implemented using the Kermeta language [5]. The transformation from a set of lattices to a UML class model is specified by three types of rules: non-relational, relational, and specialization. Figure 13 shows the rules used for the treatment of our example. At the LHS of the arrows are the patterns of the lattices and at the RHS, two views on generated UML static models are given: the model as an instance of the UML metamodel and the equivalent model in the concrete UML syntax.

The non-relational rules are the following:

- Concepts of the lattice associated with metaclass `M` give rise to UML instances of `M`; for example, concepts of lattice $\mathcal{L}_{Class}$ are interpreted as classes while concepts of lattice $\mathcal{L}_{Property}$ are interpreted as properties (more particularly attributes in the restricted metamodel we use). In rules R1 and R2 of Figure 13, concept C_i of the lattice $\mathcal{L}_{Class}$ is transformed into a UML class; while concept C_j of the lattice $\mathcal{L}_{Property}$ is transformed into a UML attribute.
- Non-relational descriptors in the intension of a concept correspond to attributes of metaclasses; for example `name` in the case of both classes and properties. In Figure 13, the names of the class generated from the concept C_i and of the attribute generated from the concept C_j come from values of descriptor `name` in concept intensions.

The generic relational rule is as follows. When a concept C_v is the value of a relation R in the intension of a concept C (i.e. when $(C, C_v) \in R$), then a link is

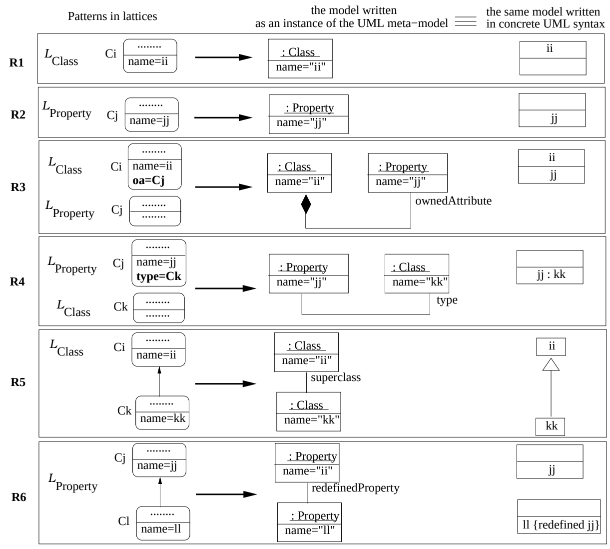


Fig. 13. Rules for the transformation from lattices to UML

created between the model element corresponding to C and the model element corresponding to C_v. The end of this link is named with the appropriate UML name corresponding to R. As an illustration, in rule R3 of Fig. 13, the intension of the concept C_i contains $ownedAttribute = C_j$ ($oa = C_j$ for short). This pattern in the lattice $\mathcal{L}_{Class}$ will be transformed into a link labelled `ownedAttribute` between the class generated from C_i and the property generated from C_j. With concrete UML syntax for class models, we obtain that class ii owns property jj. The principle is the same for rule R4.

Specialization in the class lattice gives rise to generalization/specialization links in the class diagram (R5 in Figure 13), and specialization in the property lattice is interpreted as `redefined` constraints between attributes (R6 in Figure 13).

To illustrate this transformation, the final lattices of our example are shown in Figure 14. As we stop at the fix point, concepts C^4_x and C^3_x can be considered as equivalent for any x. The refactored class diagram proposed in Figure 1(b) is obtained as follows. We first examine class lattice. Concept C^4_{BATA} is transformed into class `BankAccount`, while Concept C^4_{BAHTC} is transformed into class `BankClient` (new names are proposed by a designer after refactoring; so far arbitrary names are generated by the transformations). Concepts C^4_{BA}, C^4_{TA}, C^4_{BAH} and C^4_{TC} are respectively transformed into classes `BA`, `TA`, `BAH` and `TC`. We can say that initial classes are re-discovered. Now let's consider the property lattice. Concept C^4_{bbabta} is transformed into attribute `balance`, factorized in class `BankAccount`. From concept $C^4_{bAlist-tAlist}$ attribute `accountList` is generated.

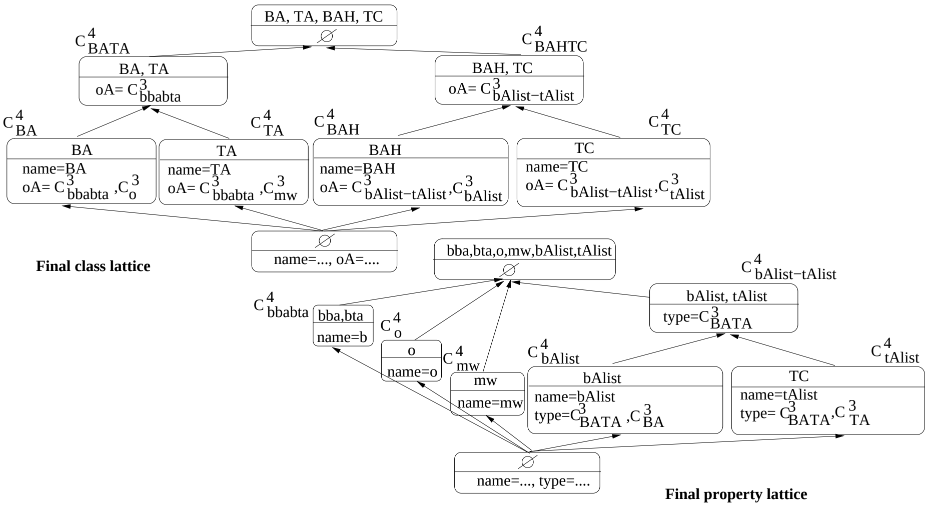


Fig. 14. The final lattices

Then we recognize initial attributes in the remaining concepts. Specialization links and `redefined` constraints stem from lattice partial order.

6 Discussion: Advantages, Limitations, and Related Work

One of the main parameters in this approach is the discovery and choice of appropriate UML elements and description of those elements to build significant abstractions. Technical description, e.g. visibility for attributes, is rather inadequate since it generates generalizations which have no semantics for the design. Nevertheless this description has to be preserved and even sometimes generalized in final step. Multiplicities are a good example: they are not interesting in the main transformation, but they should be re-injected in the last UML model and even generalized.

One advantage is that the current specification of the approach is easily transposable to a large set of UML elements (associations, parameters, operations, etc.). We are currently working on specifying the entire process at a higher level (M3) in the four-layered metamodeling hierarchy. This would allow to better demonstrate that first and second transformations can be done for any other modeling language, just by specifying which are entities, attributes and relations.

Another feature of our approach is that the technique will be useful if the designer can easily fine-tune the selection of those entities, attributes and relations, beyond traceability issues. The designer should be given the possibility to choose the subset of UML elements he considers as relevant for a RCA application.

A last problem is determining a reasonable bound on the iteration number, since at each iteration, abstractions are further and further from the model elements which have triggered the generalization. Too abstract elements can be less useful.

When specifying the metamodels and implementing the transformations, the choice of the Kermeta language appeared as a good choice. Indeed, its compatibility with MOF made it possible to use a single language for the whole implementation and its imperative syntax made the transformation implementation easy enough, whereas expressing them with a declarative syntax would have been very difficult. FCA has been used in various software engineering tasks, as shown in surveys like [9,10]. Conceptual model construction has been studied with the support of FCA, as database schema construction [11,12], class hierarchy construction or restructuring using class features [2,3,13,14,15,16] or based on feature usage [4]. Nevertheless, FCA usage has not yet been studied in the context of Model Driven Engineering, even if several contributions were proposed concerning model refactoring. A survey of software refactoring can be found in [17], and a section is dedicated 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 [18]. 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 [19]).

7 Conclusion

This paper presents an approach to automatically detect and build relevant abstractions in a UML class model. This method is founded on Relational Concept Analysis, an extension of Formal Concept Analysis. It proceeds by successive applications of model transformations, based on different metamodels (UML 2.0, context, and lattice metamodels) and implemented with the model-oriented language Kermeta. The application of our approach results in introducing abstractions for classes (with specialization links), attributes, methods and so on, in a class model. In fact, any kind of model element can be abstracted, but only a few of them lead to relevant abstractions. Future work will consist in proposing to the final users the way to parameterize the application by the metamodel elements. We are also working on defining our model transformations totally independently from the UML 2.0 metamodel, to be able to apply it on any entry metamodel. Finally, we are starting a collaboration with natural language experts to improve the refactored class diagram with relevant names for the abstractions, and to resolve problems due to synonymy, homonymy and hyperonymy.

Acknowledgements: Gabriela Arévalo gratefully acknowledges the financial support of the Swiss National Foundation for the Project: "Advanced Object-Oriented Reverse Engineering using Formal Concept Analysis" SNF Project No. PBBE2-111194. We also acknowledge the useful comments from the anonymous reviewers of this paper.

References

1. Ganter, B., Wille, R.: Formal Concept Analysis, Mathematical Foundations. Springer, Berlin (1999)
2. Godin, R., Mili, H.: Building and maintaining analysis-level class hierarchies using Galois lattices. In: Proc. of OOPSLA'93, Washington (DC), USA. (1993) 394–410
3. Dicky, H., Dony, C., Huchard, M., Libourel, T.: On Automatic Class Insertion with Overloading. In: Special issue of Sigplan Notice, Proc. of OOPSLA'96. (1996) 251–267
4. Snelting, G., Tip, F.: Understanding class hierarchies using concept analysis. ACM Transactions on Programming Languages and Systems **22**(3) (2000) 540–582
5. Triskell project (IRISA): The Metamodeling Language Kermeta. http://www.kermeta.org (2006)
6. Dao, M., Huchard, M., Hacène, M.R., Roume, C., Valtchev, P.: Improving Generalization Level in UML Models: Iterative Cross Generalization in Practice. In: ICCS'04. Volume 3127 of Lecture Notes in Computer Science., Springer (2004) 346–360
7. OMG: UML version 2.0. http://www.omg.org/technology/documents/formal/-uml.htm (2006)
8. Steel, J., Jézéquel, J.M.: Model typing for improving reuse in model-driven engineering. In: Proceedings of MODELS/UML'2005. (2005) 84–96
9. Tilley, T., Cole, R., Becker, P., Eklund, P.: A survey of formal concept analysis support for software engineering activities. In: Proc. of the First International Conference on Formal Concept Analysis - ICFCA'03, Springer-Verlag (2003) 250–271
10. Arévalo, G.: High-Level Views in Object-Oriented Systems using Formal Concept Analysis. PhD thesis, Software Composition Group, University of Bern (2004)
11. Yahia, A., Lakhal, L., Cicchetti, R., Bordat, J.: iO2 - An Algorithmic Method for Building Inheritance Graphs in Object Database Design. In: Proc. of the 15th International Conf. on Conceptual Modeling ER'96. Volume 1157. (1996) 422–437
12. Andonnoff, E., Sallaberry, C., Zurfluh, G.: Interactive design of object oriented databases. In: Proc. of CAISE'92. Volume 593 of LNCS., Springer-Verlag (1992) 128–146
13. Cook, W.: Interfaces and Specifications for the Smalltalk-80 Collection Classes. In Paepcke, A., ed.: Proceedings of the 10th OOPSLA, ACM Press (1992) 1–15
14. Moore, I.: Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In: Proceedings of OOPSLA'96, San Jose (CA), USA. (1996) 235–250
15. Chen, J.B., Lee, S.C.: Generation and reorganization of subtype hierarchies. Journal of Object Oriented Programming **8**(8) (1996) 26–35
16. Si-Said Cherfi, S., Lammari, N.: Towards and Assisted Reorganization of Is-A Hierarchies. In: Proc. of Object-Oriented Information Systems, Springer-Verlag (2002) 536–548
17. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Transactions on Software Engineering **30**(2) (2004) 126–139
18. Sunyé, G., Pollet, D., Le Traon, Y., Jézéquel, J.M.: Refactoring UML models. In: Proc. Unified Modeling Language Conf. (2001) 134–148
19. Tokuda, L., Batory, D.: Automated software evolution via design pattern transformations. In: Proc. of the Int'l Symp. on Applied Corporate Computing. (1995)